Visual C++ 音频/视频处理技术及工程实践

路锦正　周冬梅　编著
飞思科技产品研发中心　监制

C/C++ 开发专家

一线音/视频开发工程师8年沉淀的经验奉献。
9大经典案例全面剖析应用开发技术，学有所用。
配套工程原型代码，快速构建用户应用程序。

随书所附光盘内容为
书中实例源文件
CD-ROM

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

专业人士 权威经典

C/C++ 开发专家

Visual C++ 音频/视频处理技术及工程实践

路锦正　周冬梅　编著
飞思科技产品研发中心　监制

電子工業出版社
Publishing House of Electronics Industry
北京·BEIJING

内容简介

本书内容包含16章，分为4篇，详细讲解了使用各种软件和平台进行音/视频多媒体编程的技术，以案例为对象展示实现过程、分析技术难点。主要内容包括 Visual C++ 2005 的开发技术、DirectSound 开发音频、DirectShow/VFW 开发视频、MMX/SSE 进行多媒体汇编编程、DM642 DSP 进行音/视频算法优化和主流视频算法 MPEG-4/H.264 的编码原理及工程实践。

本书系统地介绍了使用 Visual C++ 2005 进行流媒体编程的基本思路和方法，采用案例为主的叙述方式，将大量的技术理论融入具体的案例剖析中。书中采用的案例均来源于作者实际开发工作，具有很好的实用价值，方便广大开发者在开发中进行参考或直接应用。

随书所附光盘包含书中实例源文件。

本书内容丰富、安排合理、工程实用性强，能够为广大数字媒体或音/视频开发人员、高等院校相关专业的课程设计、毕业设计提供参考，同时也可以作为科研单位、企业进行流媒体开发的技术指导用书。

未经许可，不得以任何方式复制或抄袭本书的部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

Visual C++音频/视频处理技术及工程实践 / 路锦正，周冬梅编著.—北京：电子工业出版社，2009.1

（C/C++开发专家）

ISBN 978-7-121-07890-3

I. V… Ⅱ.①路…②周… Ⅲ.C 语言－程序设计 Ⅳ.TP312

中国版本图书馆 CIP 数据核字（2008）第 184468 号

责任编辑：杨　鸲

印　　刷：北京智力达印刷有限公司

装　　订：三河市鹏成印业有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱　　邮编：100036

开　　本：787×1092　1/16　印张：41.25　　字数：1068.8 千字

印　　次：2009 年 1 月第 1 次印刷

印　　数：4 000 册　　　定价：76.00 元（含光盘 1 张）

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

Visual C++ 2005 对 C++标准几乎无缝支持，它集成的开发环境（IDE）相比较早期的版本更强大、功能更丰富，可以大大提高开发效率。Visual C++ 2005 是 Visual Studio 2005 集成环境，以后转 C#、ASP.NET、VB.NET 更容易；Visual C++ 2005 的 MSDN 与开发环境是集成的，使用在线帮助更简单；Visual C++ 2005 的 MFC 编程更方便、强大；Visual C++ 2005 支持 Visual C++.NET。

Visual C++ 2005 使用易跨平台、通用性强的 C++或 C 语言编程，几乎支持所有的开发技术：网络、数据库、图形图像、多媒体等应用开发，应用程序、静态/动态链接库、驱动程序等全系列程序开发。目前市面上关于 Visual C++ 6.0 的开发书籍种类繁多，也有众多的技术人员从事 Visual C++ 6.0 的编程开发，关于 Visual C++ 6.0 的技术网站或论坛也是数不胜数。Visual C++ 6.0 的通用性、普适性可见一斑。而 Visual C++ 6.0 的升级版 Visual C++ 2005 的有关技术书籍却很少，关于 Visual C++ 2005 音/视频编程的书籍，目前市面上更是没有。

音/视频处理的 PC 软件工具有 DirectSound、DirectShow、VFW 等，这些 SDK 都可以在 Visual C++ 2005 环境下编程。VC 开发结合 DSP 编程是进行音/视频开发的典型路线。主流的视频算法 MPEG-4/H.264 实现平台有 VC 和 DSP，对于算法实用工程本书选择最优秀的开源代码 Xvid 和 X264。本书就是在 Visual C++ 2005 这一广为软件开发者选用的平台下，进行音/视频的捕获采集、流媒体数据的播放、音/视频数据的编/解码，讲解使用 MMX/SSE 多媒体汇编语言和 DSP 进行视频编/解码等。

本书精选了实用工程中的原型案例，结合源代码分析音/视频编程的思路和难点。用户既可以直接使用这些案例，也可以基于此继续丰富功能，编写出自己的音/视频程序。所有案例的源代码和使用说明均附在光盘上以供读者使用。

主要内容

本书全面而细致地讲解 Visual C++ 2005 进行音/视频开发、算法优化的技术，全书分为 4 篇，共 16 章。具体内容如下所示。

第 1 篇开发基础，包括第 1 章～第 2 章。首先对数字音/视频的基本概念进行了概述，同时也介绍了音/视频处理技术、音/视频处理平台、音/视频开发工具，以及音/视频技术在日常生活中的应用。对于刚从事音/视频开发的新手而言，了解和掌握使用 Visual C++ 2005 开发工具进行音/视频开发也是非常重要的。接着详细讲解了 Visual C++ 2005 的开发基础，从它的基本概念和操作入手，详细阐述了应用程序向导、MFC 编程基础、基于对话框的编程、常用控件的使用，各知识点按照由浅入深的方法进行讲述。本部分是音/视频开发的基础知识，是读者学习音/视频开发必须熟悉和掌握的内容。

第 2 篇音频开发，包括第 3 章～第 6 章。本部分描述了 DirectSound 音频开发基础，并通过第 4 章~第 6 章的实例对 DirectSound 开发技术进行讲解。由于 DirectSound 在音频开发中具有优势，此部分首先对 DirectSound 的基本结构、具体使用方法、开发环境配置，以及音频应用程序的调试技术进行了详解。接着通过具体案例详细介绍了基于 Visual C++ 2005 的音频开发技巧和方法，每个实例都按照功能分析、设计思路、编程原理和实例详解 4 个步骤进行阐

述。本部分是音频开发的重要内容，通过此部分的学习，读者可以理解和掌握 DirectSound 开发的步骤和方法。同时读者也可以试着设计出自己的音频应用程序了！

第 3 篇**视频开发，包括第 7 章～第 10 章。**本篇详细讲解了微软 DirectShow SDK 和 VFW 的视频开发技术。介绍了 DirectShow SDK 的获取和安装、Visual C++ 2005 开发环境配置及常见开发问题、DirectShow 的工作原理、滤波器（Filters）构成及分类、滤波器链表工作原理、事件通知机制、常用的 DirectShow SDK 接口等。使用 DirectShow SDK 可以快速构建播放多媒体文件、捕获采集音/视频的应用程序。该部分以两个案例分析 DirectShow SDK 的开发流媒体回放、音/视频采集过程。同时还介绍了 VFW 的视频采集功能，以案例下的两个应用 VFW 的压缩管理器（ICM）和用户算法分别实现视频编码，阐述视频通用平台。通过本篇的学习，读者可以掌握视频开发的工具平台，基于本部分的案例设计出功能更丰富的视频应用程序。

第 4 篇 **编/解码技术，包括第 11 章～第 16 章。**本篇是高级应用篇，编/解码是音/视频处理的重要方法。首先概述了数字音/视频的编/解码方法，回顾了当前的编/解码技术标准及应用领域和技术指标等，总结了算法开源资源，展望了编/解码产业。接着介绍了多媒体计算机的 MMX/SSE 多媒体汇编指令，并剖析了指令特点；通过案例详细介绍 MMX 和 SSE 进行算法优化的过程，并展示技术效果；使用汇编指令优化视频编/解码算法，并列举了开发中的常见问题。然后展示使用 DSP（DM642）实现视频编/解码，包括 CCS 的安装、开发过程，使用 C 语言、线性汇编来设计、优化算法的思路，并以创建库工程和应用工程为对象分析开发常见问题。最后以 3 个实用工程案例详细介绍了视频算法开发和系统应用，包括开源代码 XviD CODEC 实现 MPEG-4 视频编/解码算法、开源代码 X264 实现 H.264/AVC 视频编码和 DVR 视频监控系统设计等案例。这部分中的案例，供读者直接参考以开发自己的视频应用，或基于这些工程设计出用户自己的程序。

本书特色

- ✧ 音/视频工程师兼培训讲师 8 年经验的倾情奉献。
- ✧ 剖析了 DirectSound、DirectShow 的功能与特性，以案例详细讲解技术开发过程。
- ✧ 重点对主流的视频算法 MPEG-4/H.264 从编码原理、工程实践、算法优化、技术效果等出发介绍最实用的开源算法 XviD CODEC 和 X264 编码。
- ✧ 内容丰富而且翔实，一书在手，Visual C++音/视频开发、算法优化无忧。
- ✧ 全程案例引导学习，目标更加明确，学习效率更高。
- ✧ 源于实践，贴近应用，案例为主，内容生动，深入浅出，通俗易懂。
- ✧ 配套代码光盘，免去烦琐输入代码的工作，提高学习效率。

读者对象

本书可作为从事 Visual C++入门开发、DirectSound 音频开发、DirectShow 视频开发、MMX/SSE 多媒体汇编编程、DSP 算法优化等开发人员的参考指南。本书既可以作为高等院校相关专业的教材，同时也可以作为科研单位、企业进行流媒体开发的技术指导用书。

本书约定

本书的插图和运行结果可能与读者实例环境中的操作界面或结果略有差别，这可能是由于操作系统平台、Visual C++、CCS 软件版本不同而导致的，本书使用 Windows XP（SP2）、Visual C++ 2005 简体中文专业版、CCS 2.20.18。在此特别说明，一切以实际环境为准。

致谢与分工

本书由路锦正、周冬梅编著，其中第 1、2 篇由周冬梅编写，第 3、4 篇由路锦正编写，全书由路锦正统稿，另外成都理工大学的朱朝飞也参与了部分章节的编写，姚新军负责前期策划与后期质量控制。本书由成都易为科技有限责任公司审校，参与编辑、排版、校对等工作的有：黄中林、张强林、王斌、万雷、王晓、吴艳、赵会春、王呼佳、赵腾伦、余松、魏雪辉、夏慧军等。感谢科胜讯数字电视（成都）有限公司的王建勤总工提供了宝贵的信息支持。感谢西南科技大学信息工程学院的张笑微教授及学院的领导，对著书工作的大力支持。在此要特别感谢家人的大力支持。另外，本书参考了网络上公开的部分资源，对这些资源的所有者、网友在此一并表示感谢。

由于时间有限，加之水平有限，书中不足之处在所难免，恳请读者批评指正。最后还需要感谢电子工业出版社的辛苦工作的老师们，他们为本书的顺利出版付出了很多辛勤劳动。

编 著 者

联系方式

咨询电话：（010）68134545　　88254160

电子邮件：support@fecit.com.cn

服务网址：http://www.fecit.com.cn　　http://www.fecit.net

通用网址：计算机图书、飞思、飞思教育、飞思科技、FECIT

目 录

第 1 篇 开发基础 …… 1

第 1 章 数字音/视频开发技术基础 …… 3

1.1 数字音/视频基本概念 …… 4

1.1.1 数字音频技术基础 …… 4

1.1.2 各种数字音频文件格式基础 …… 4

1.1.3 视频的相关基础概念 …… 8

1.1.4 常见的视频文件格式 …… 8

1.2 数字音/视频数据的编/解码技术 …… 10

1.2.1 音频数据的编/解码 …… 11

1.2.2 视频数据的编/解码 …… 13

1.2.3 音/视频编码的相关标准 …… 13

1.3 音/视频处理平台的选择 …… 15

1.3.1 个人多媒体计算机 …… 15

1.3.2 嵌入式处理器 ARM …… 16

1.3.3 数字媒体处理器 DSP …… 16

1.4 音/视频开发的典型应用 …… 16

1.4.1 个人消费类电子产品 …… 16

1.4.2 公共领域的音/视频应用 …… 18

1.5 常用音/视频开发工具 …… 21

1.5.1 DirectSound 音频开发 …… 21

1.5.2 DirectShow 视频开发 …… 21

1.5.3 DSP CCS 算法开发 …… 22

1.6 本章小结 …… 23

第 2 章 Visual C++ 2005 开发基础 …… 25

2.1 Visual C++ 2005 新增特性 …… 26

2.1.1 句柄（Handles） …… 26

2.1.2 类型的声明 …… 26

2.1.3 对代码编辑的改进 …… 27

2.2 VC 能做的事情 …… 27

2.2.1 生成传统的控制台应用程序 …… 27

2.2.2 生成基于 MFC 的应用程序 …… 32

2.2.3 生成静态链接库 …… 40

2.2.4 生成动态链接库 …… 41

2.3 利用向导完成基本开发 …… 46

2.3.1 生成应用程序 …… 46

2.3.2 生成应用库 LIB/DLL …… 53

2.4 MFC 编程基础 …… 56

2.4.1 Windows 消息与事件 …… 56

2.4.2 消息与事件的响应 …… 63

2.4.3 常用消息 …… 66

2.5 基于对话框的编程 …… 68

2.5.1 创建和编辑对话框 …… 68

2.5.2 对话框种类 …… 75

2.6 常用控件使用技巧 …… 79

2.6.1 按钮类控件的使用 …… 80

2.6.2 文本可编辑类控件的使用 …… 81

2.6.3 静态类控件的使用 …… 82

2.6.4 其他控件 …… 83

2.7 Visual 2005 开发技术资源 …… 84

2.7.1 MSDN 在线帮助 …… 84

2.7.2 丰富的网络资源 …… 86

2.8 Visual 2005 开发的常见问题 …… 86

2.8.1 如何使用 volatile 和 const_cast …… 86

2.8.2 如何构造 unicode 版本的程序 …… 87

2.8.3 如何使用.def 文件 …… 87

2.8.4 如何正确编写 Win32 DLL …… 87

2.8.5 如何编写和使用资源 DLL …… 88

2.8.6 如何实现对话框的数据交换 …… 88

2.9 本章小结 …… 89

第 2 篇 音频开发 …… 91

第 3 章 DirectSound 开发基础 …… 93

3.1 Microsoft DirectSound SDK …… 94

3.1.1 比较 DirectSound 与 DirectMusic …… 94

3.1.2 DirectSound SDK 能做什么 …… 95

3.1.3 如何获取 DirectSound SDK …… 96

3.1.4 DirectSound SDK 的基本结构 …… 98

3.1.5 DirectSound SDK 安装 …… 100

3.1.6 选择最适合用户的 DirectSound SDK …… 100

3.1.7 用 DirectSound 进行编程的必要步骤 …… 102

3.2 DirectSound 的详细使用方法 …… 103

3.2.1 DirectSound 设备对象 …… 104

3.2.2 DirectSound 缓冲区对象 …… 107

3.2.3 使用 WAV 数据 …… 114

3.2.4 使用音效 …… 116

3.3 开发环境配置 …… 118

3.3.1 Visual C++基本开发环境配置 …… 119
3.3.2 DirectSound 应用程序的开发环境配置 …… 119
3.3.3 DirectSound Filter 开发环境配置 …… 121
3.4 调试音频处理程序 …… 125
3.4.1 Visual C++调试器 …… 125
3.4.2 DirectSound 中常用的调试技术 …… 128
3.5 DirectSound 开发的常见问题 …… 130
3.6 本章小结 …… 131
第 4 章 打造自己的音频播放器 …… 133
4.1 系统设计和分析 …… 134
4.2 缓冲区和通知基础 …… 135
4.2.1 缓冲区的概念 …… 135
4.2.2 缓冲区的通知 …… 136
4.3 播放文件 …… 137
4.3.1 创建 DirectSound 对象 …… 137
4.3.2 创建播放缓冲区 …… 138
4.3.3 调用方法播放声音 …… 145
4.3.4 结束播放 …… 146
4.4 系统效果演示 …… 146
4.4.1 新建基于对话框的程序框架 …… 147
4.4.2 添加控件及变量 …… 147
4.4.3 封装 CDirectSound 类实现 DirectSound 声音播放 …… 148
4.4.4 与对话框界面相关操作实现 …… 156
4.4.5 程序运行结果 …… 159
4.5 DirectSound 开发常见问题 …… 160
4.6 本章小结 …… 161
第 5 章 语音 3D 特效器制作 …… 163
5.1 系统设计和分析 …… 164
5.2 DirectSound 3D 缓冲区 …… 164
5.2.1 获取 3D 缓冲区对象 …… 165
5.2.2 最大距离和最小距离 …… 166
5.2.3 处理模式 …… 167
5.2.4 声音圆锥 …… 167
5.3 DirectSound 3D 虚拟听众 …… 168
5.3.1 获得 3D 听众 …… 168
5.3.2 听众的空间参数 …… 169
5.3.3 距离因子与 Doppler 效应 …… 171
5.3.4 滚降因子与延迟设置 …… 173
5.4 系统效果演示 …… 174
5.4.1 具体的实现步骤 …… 174
5.4.2 程序的实现代码 …… 176
5.4.3 程序运行结果 …… 186
5.5 DirectSound 3D 开发常见问题 …… 187
5.6 本章小结 …… 188
第 6 章 对拾音设备录制自己的语音文件 …… 189
6.1 系统分析和设计 …… 190
6.2 管理捕获缓冲区 …… 191
6.2.1 创建捕获缓冲区 …… 191
6.2.2 启动缓冲区 …… 197
6.2.3 封锁缓冲区 …… 197
6.2.4 解锁缓冲区 …… 198
6.2.5 中止捕获 …… 199
6.3 系统效果演示 …… 199
6.3.1 声音类的封装 …… 199
6.3.2 声音管理函数的实现 …… 202
6.3.3 程序运行结果 …… 214
6.4 DirectSound 录制语音文件开发常见问题 …… 215
6.5 本章小结 …… 216
第 3 篇 视频开发 …… 217
第 7 章 DirectShow 开发基础 …… 219
7.1 Microsoft DirectShow SDK …… 220
7.1.1 DirectX 及成员组成 …… 220
7.1.2 DirectShow SDK 能做什么 …… 221
7.1.3 如何获取 DirectShow SDK …… 221
7.1.4 安装 DirectShow SDK …… 222
7.2 开发环境配置 …… 224
7.2.1 生成 DirectShow SDK 开发库 …… 224
7.2.2 Visual C++开发环境配置 …… 229
7.2.3 常见问题解析集锦 …… 232
7.3 DirectShow SDK 基本架构 …… 234
7.3.1 DirectShow SDK 总体架构 …… 234
7.3.2 DirectShow SDK 简单应用程序案例 …… 236
7.3.3 滤波器链表 Filer Graph 及各种组件 …… 244
7.3.4 构建滤波器链表（Building Filer Graph） …… 252
7.3.5 滤波器链表中的数据流动 Data Flow …… 257

7.3.6 DirectShow 中的事件通知机制 ··· 261
7.3.7 常用 DirectShow SDK 接口 ··· 267
7.4 DirectShow SDK 常用开发案例 ··· 267
7.4.1 通用开发流程 ··· 268
7.4.2 系统初始化 ··· 268
7.4.3 媒体播放方法 ··· 268
7.4.4 消息处理方法 ··· 270
7.4.5 视频采集技术 ··· 271
7.4.6 音频采集案例 ··· 282
7.5 本章小结 ··· 293
第 8 章 打造自己的媒体播放器 ··· 295
8.1 系统分析与设计 ··· 296
8.1.1 FilterGraph 结构设计 ··· 297
8.1.2 GraphEdit 模拟实现 ··· 297
8.1.3 界面设计 ··· 299
8.2 实现媒体控制类 ··· 307
8.2.1 CDXGraph 类初始化 ··· 308
8.2.2 创建 Graph 滤波器链表 ··· 309
8.2.3 图像窗口设计 ··· 311
8.2.4 媒体播放控制 ··· 312
8.2.5 全屏显示及抓图存盘实现 ··· 315
8.3 实现一个简单的媒体播放器 ··· 317
8.4 升级你的播放器 ··· 321
8.4.1 添加控制功能 ··· 321
8.4.2 添加拖放功能 ··· 325
8.4.3 添加音量调节功能 ··· 326
8.4.4 添加欢迎背景图片 ··· 328
8.5 系统效果展示 ··· 328
8.6 如何播放更多的文件格式 ··· 329
8.7 本章小结 ··· 330
第 9 章 自制 DV 实现视频采集 ··· 331
9.1 系统分析与设计 ··· 332
9.1.1 采集设备的枚举 ··· 332
9.1.2 使用 Capture Graph Builder ··· 335
9.1.3 采集参数的设置 ··· 336
9.2 使用经典采集技术实现视频捕获 ··· 338
9.2.1 GraphEdit 模拟实现 ··· 338
9.2.2 视频捕获类 CCaptureClass 的实现 ··· 340
9.2.3 界面设计 ··· 345
9.3 使用 VMR 技术实现视频图像捕获 ··· 350
9.3.1 VMR 技术基础 ··· 350
9.3.2 GraphEdit 模拟实现视频捕获预览 ··· 350
9.3.3 视频图像捕获类 CVMR_Capture 的实现 ··· 353
9.3.4 界面设计 ··· 361
9.4 系统效果演示 ··· 366
9.4.1 实例一　系统效果演示 ··· 367
9.4.2 实例二　系统效果演示 ··· 368
9.5 本章小结 ··· 369
第 10 章 VFW 技术实现视频处理通用平台 ··· 371
10.1 VFW 开发技术流程分析 ··· 372
10.1.1 技术概述 ··· 372
10.1.2 VFW 采集开发流程图 ··· 373
10.2 使用 VFW 实现视频捕获和预览 ··· 373
10.2.1 建立单文档应用程序 ··· 373
10.2.2 创建视频窗口 ··· 375
10.2.3 设计回调函数 ··· 376
10.2.4 视频图像显示设置 ··· 378
10.2.5 捕获预览视频 ··· 379
10.2.6 配置视频格式及图像参数 ··· 381
10.3 使用系统插件实现视频编/解码 ··· 382
10.3.1 使用系统视频编/解码插件 ··· 383
10.3.2 视频编码码流写 AVI 文件 ··· 385
10.3.3 关闭平台 ··· 387
10.4 使用 XviD CODEC 算法实现视频编/解码 ··· 388
10.4.1 生成 XviD 算法静态库（编译过程） ··· 388
10.4.2 实现 XviD CODEC 视频编码算法函数 ··· 389
10.4.3 实现 XviD CODEC 视频解码算法函数 ··· 393
10.4.4 使用 XviD CODEC 编/解码算法 ··· 395
10.5 平台应用效果展示 ··· 398
10.6 本章小结 ··· 399
第 4 篇 编/解码技术 ··· 401
第 11 章 纵览音/视频编/解码技术 ··· 403
11.1 数字音/视频基础 ··· 404
11.2 音/视频编/解码及方法 ··· 406

11.2.1 音频编码方法 …… 406
11.2.2 视频编码方法 …… 410
11.3 编/解码技术标准 …… 417
11.3.1 静态图像编码标准 …… 418
11.3.2 MPEG-4/H.264 视频算法 …… 424
11.3.3 AVS 国产视频编码标准 …… 430
11.4 编/解码产业及发展 …… 431
11.4.1 编/解码资源一览 …… 431
11.4.2 编/解码发展前景 …… 432
11.5 本章小结 …… 433
第 12 章 使用 MMX/SSE 2 进行多媒体编程 …… 435
12.1 MMX/SSE 2 技术基础 …… 436
12.2 MMX/SSE 2 指令剖析 …… 439
12.2.1 MMX 媒体扩展指令 …… 439
12.2.2 MMX 程序设计 …… 444
12.2.3 SSE/SSE 2 媒体扩展指令 …… 455
12.2.4 SSE 程序设计详细解析 …… 457
12.3 使用 MMX/SSE 2 进行音/视频开发 …… 463
12.3.1 开发前技术准备 …… 464
12.3.2 MMX/SSE 2 视频编/解码编程 …… 465
12.3.3 使用 VC 调试 MMX/SSE 2 程序 …… 470
12.4 MMX/SSE 2 开发常见问题 …… 472
12.4.1 制订计划 …… 472
12.4.2 哪部分代码可使用 MMX 技术改进 …… 473
12.4.3 代码是浮点型还是整型 …… 473
12.4.4 EMMS 准则 …… 474
12.4.5 CPUID 的检测 MMX 技术的用法 …… 474
12.4.6 数据对齐 …… 474
12.4.7 数据安排 …… 475
12.4.8 应用程序最后的调整 …… 476
12.5 本章小结 …… 476
第 13 章 用 DM642 实现视频编/解码技术 …… 477
13.1 数字媒体处理器 TMS320DM642DSP 概述 …… 478
13.2 DSP 软件开发环境 CCS …… 481
13.2.1 安装 CCS …… 481
13.2.2 CCS 主要部件 …… 482
13.2.3 使用 CCS 优化工具实现算法优化 …… 488
13.3 用 C 语言进行视频算法框架编程 …… 492
13.3.1 C 编程规则和参考 …… 492
13.3.2 DSP 关键字与 CMD 文件使用 …… 493
13.3.3 算法系统资源剖析 …… 495
13.4 DM642 实现视频算法优化 …… 497
13.4.1 并行算法指令和建立软件流水 …… 497
13.4.2 使用 intrinsic 指令完成核心模块的优化 …… 499
13.4.3 使用 DSP 线性汇编优化核心模块 …… 500
13.4.4 使用 Cache 技术实现算法优化 …… 507
13.4.5 使用乒乓式 EDMA 实现算法优化 …… 508
13.5 DM642 优化视频算法常见问题 …… 513
13.6 本章小结 …… 518
第 14 章 XviD CODEC 实现 MPEG-4 编/解码 …… 519
14.1 MPEG-4 编/解码概述 …… 520
14.1.1 基于对象的 MPEG-4 视频编码 …… 520
14.1.2 XviD 格式文件播放 …… 520
14.2 XviD CODEC 编/解码分析 …… 521
14.2.1 MPEG-4 编/解码设计与剖析 …… 521
14.2.2 MMX/SSE 2 实现 XviD CODEC …… 549
14.2.3 DM642 下 XviD CODEC 优化 …… 553
14.3 运行 XviD CODEC 系统 …… 559
14.3.1 YUV 原始视频数据及其显示 …… 559
14.3.2 VC 平台下编译和运行 XviD CODEC …… 559
14.3.3 CODEC 在 DSP 下软仿真和硬仿真 …… 562
14.4 系统效果展示 …… 562
14.5 本章小结 …… 563

第 15 章　X264 实现 H.264/AVC 视频编码……565
15.1　H.264/AVC 编码概要……566
15.1.1　开源代码工程管理软件……566
15.1.2　获取开源算法工程……566
15.1.3　H.264/AVC 新特性……567
15.2　X264 视频编码分析……568
15.2.1　H.264/AVC 关键技术要点……569
15.2.2　X264 视频编码设计与剖析……577
15.2.3　X264 核心模块 MMX/DSP 汇编优化……599
15.3　运行 X264 编码系统……606
15.3.1　VC 平台下编译和运行 X264……606
15.3.2　JM 验证 X264 码流……610
15.3.3　ffmpeg 中的 H.264 视频解码……611
15.4　系统效果展示……611
15.4.1　X264 实现 H.264/AVC 视频编码……611
15.4.2　ffmpeg-h264-vc 工程实现 H.264 视频解码……612
15.5　本章小结……613
第 16 章　打造自己的 DVR 监控系统……615
16.1　DVR 监控系统分析与设计……616
16.1.1　监控系统分类……616
16.1.2　监控系统组成……617
16.1.3　基于压缩板卡的 SDK 软件开发包……619
16.2　DVR 监控系统主界面软件设计……621
16.2.1　对话框应用程序开发……621
16.2.2　位图按钮设计……622
16.3　模拟 DVR 视频采集……636
16.3.1　基于 DirectShow 的视频采集回放……636
16.3.2　基于 DirectShow 的影音文件回放……638
16.4　系统效果展示……643
16.5　本章小结……644
附录 A　旧版 VC 升级到 VC 2005 程序安全的 10 点注意事项……645
附录 B　开发常见问题……647

第 1 篇

开发基础

本篇首先对数字音/视频的基本概念进行了概述，同时也介绍了音/视频处理技术、音/视频处理平台、音/视频开发工具，以及音/视频技术在日常生活中的应用。对于刚从事音/视频开发的新手而言，了解和掌握使用 Visual C++ 2005 开发工具进行音/视频开发也是非常重要的。接着详细讲解了 Visual C++ 2005 的开发基础，从它的基本概念和操作入手，详细阐述了应用程序向导、MFC 编程基础、基于对话框的编程及常用控件的使用，各知识点按照由浅入深的方法进行讲述。本篇是音/视频开发的基础知识，是读者学习音/视频必须熟悉和掌握的内容。

- 第 1 章　数字音/视频开发技术基础
- 第 2 章　Visual C++ 2005 开发基础

第1章

数字音/视频开发技术基础

本章主要介绍数字音/视频开发技术的相关基础知识，通过本章的学习，读者能对多媒体的数字音/视频技术的开发技术和应用有一个整体的了解，为读者后续章节的学习打好基础。

本章重点

- 数字音/视频基本概念
- 数字音/视频数据的编/解码技术
- 音/视频处理平台的选择
- 音/视频开发的典型应用
- 常用音/视频开发工具

PDG

1.1 数字音/视频基本概念

随着大量信息的音/视频数据被广泛使用，比如常见的 VCD、DVD 等都是将大量的音/视频数据经过处理后提供给用户使用的。音/视频相关技术在一些领域也得到了很好的应用，如语音识别系统、视频会议系统、远程视频监控系统等。

特别是近年来，多媒体以其震撼的音效、细腻的画面和逼真的动画等效果而引起了人们的非常关注，并应用于网络，具有广阔的市场前景。

由于多媒体的应用领域非常广，到目前为止人们对它还没有一个非常准确的定义。目前，多媒体只是在听觉、视觉方面有众多的表现形式。

多媒体技术是将所有文本（字符）、图形、图像、声音、音乐、视频、动画这些媒体形式集成起来，使我们能以更加自然的方式使用信息和计算机进行交流，且使表现的信息图、文、声并茂。因此多媒体技术是计算机集成、音频视频处理集成、图像压缩技术、文字处理和通信等多种技术的完美结合。

1.1.1 数字音频技术基础

大家网 TopSage.com

数字音频技术涉及的关键技术有：A/D 和 D/A 转换、压缩编码技术、数字信号处理技术。数字音频系统的性能在多大程度上逼近信号所确定的理论界限，取决于 A/D、D/A 转换器的性能。对音频质量而言，音频信号通过 A/D 和 D/A 变换后越接近原始的模拟音质就越好。而采样频率、量化比特率、量化噪声和频率响应是衡量 A/D 转换器性能的几个重要参数。为了提高数字音频的质量，需要提高采样频率和量化精度。随着采样频率的通过，量化比特率的不断增加，音频信号数据量的不断扩大，不方便存储与传输，因此压缩编码技术也就成为了数字音频技术的重中之重。目前大部分压缩技术属于有损压缩，压缩技术的关键指标——压缩比的提高以牺牲音频品质为代价，也就是说不能无限制地提高压缩比以满足网络传输的需求，设计必须在压缩比与音频质量之间做出折中。设计音频产品时，应该明确该产品的使用场合。在民用或对音质要求不高的场合下，可以采用压缩比较高的音频压缩技术，如常见的 MP3、WMA、RM 等压缩技术；而在专业领域中，为了保证传输过程的实时性，不能采用任何压缩技术，而是直接使用采样后的原始数据，从而保留声音原汁原味的音频品质。

1.1.2 各种数字音频文件格式基础

音频文件格式专指存放音频数据的文件的格式，存在多种不同的格式。

1）PCM 编码

PCM 脉冲编码调制是 Pulse Code Modulation 的缩写。我们常见的 Audio CD 就采用了 PCM 编码，一张光盘只能容纳 72 分钟的音乐信息。

特 点	最大优点是音质好，但最大的缺点就是体积大
适用于	在数字音响中普遍采样 PCM 编码方式

2）WAV 格式

WAV 是微软公司早期开发一种音频文件格式，它符合 PIFF（Resource Interchange File Format，资源交互文件格式）文件规范，用于保存 Windows 平台的音频信息资源，被 Windows 平台及其应用程序所支持。所有的 WAV 都有一个文件头。WAV 对音频流的编码没有硬性规定，除了 PCM 之外，还有几乎所有支持 ACM 规范的编码都可以为 WAV 的音频流进行编码。

特　点	音质非常好，被大量软件所支持
适用于	多媒体开发、保存音乐和音效素材

3）MP3 格式

MP3 作为目前最为普及的音频压缩格式，为大家广泛接受，它的全称是 Moving Picture Experts Group Audio Layer III。简单地说，MP3 就是一种音频压缩技术，由于这种压缩方式的全称为 MPEG Audio Layer 3，所以人们把它简称为 MP3。1993 年由德国 Fraunhofer IIS 研究院和汤姆生公司合作发展成功。MP3 是利用 MPEG Audio Layer 3 的技术，将音乐以 1:10 甚至 1:12 的压缩率压缩成容量较小的 file，换句话说，能够在音质丢失很小的情况下把文件压缩到更小的程度。MP3 编码技术经历了两次重大技术改进。

- VBR：VBR（Variable Bitrate，动态数据速率）技术可以让 MP3 文件的每一段甚至每一帧都有单独的 Bitrate，这样做的好处就是在保证音质的前提下最大程度地限制了文件的大小。这种技术的优越性是显而易见的，但要运用确实是一件难事，因为这要求编码器知道如何为每一段分配 Bitrate，这种技术对没有波形分析的编码器而言如同虚设。正是如此，VBR 技术并没有一出现就显得光彩夺目。
- 听觉模型的导入：专家们通过长期的声学研究，发现人耳存在遮蔽效应。声音信号实际是一种能量波，在空气或其他媒介中传播，人耳对声音能量的多少即响度或声压最直接的反应就是听到这个声音的大小，我们称它为响度，表示响度这种能量的单位为分贝（dB）。即使是同样响度的声音，人们也会因为它们的频率不同而感觉到声音的大小不同。人耳最容易听到的就是频率为 4000Hz 的声音，不管频率是否增高或降低，即使是在响度相同的情况下，大家都会觉得声音在变小。但响度降到一定程度时，人耳就听不到了，每一个频率都有着不同的值，如图 1-1 所示是人耳可听到的最小响应曲线图。

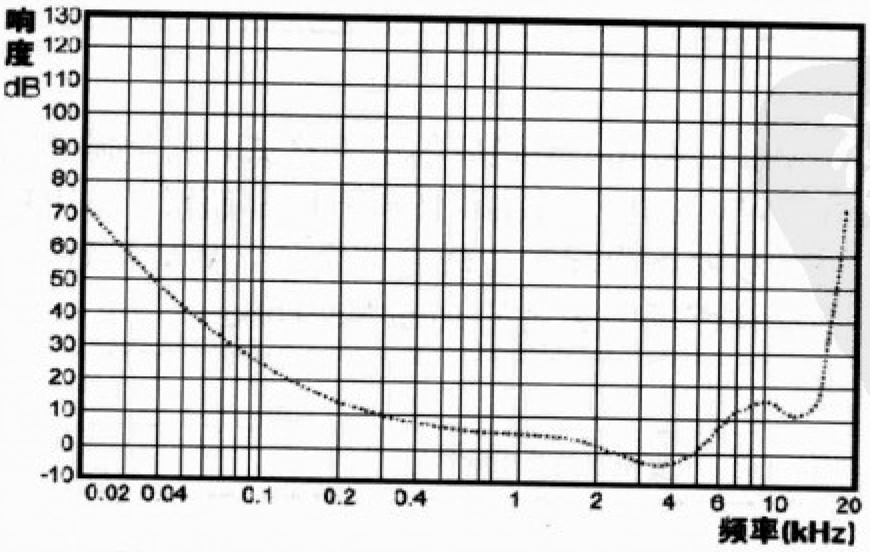


图 1-1　人耳可以听到的最小响应曲线图

可以看到这条曲线基本成一个 V 字型，当频率超过 15000Hz 时，人耳会感觉到声音很小，

向任何人提供完全免费的多媒体编码方案。OGG 的信念就是 OPEN！FREE！Vobis 则是某音乐作品中一个人物的名字，这里只是借用它来代表 OGG。Vobis 这个词汇成为了 OGG 项目中音频编码的正式命名。

特　点	可以用比 MP3 更小的码率实现比 MP3 更好的音质，高中低码率下均具有良好的表现
适用于	用更小的存储空间获得更好的音质（相对 MP3）

6）RA（rm）格式

RA 格式是 Real Networks 公司所开发的一种新型流式音频 Real Audio 文件格式。这是网上接触得非常多的一种格式，大部分音乐网站的在线试听都采用了 Real Audio，这种格式完全针对的就是网络上的媒体市场，支持非常丰富的功能。最大的优点就是这种格式可以根据听众的带宽来控制自己的码率，在保证流畅的前提下尽可能提高音质。

特　点	码率可控，支持多种音频编码，可边读边放
适用于	大部分音乐网站及互联网

7）MPC 格式

MPC 格式以前称为 MP+，它的音质优势主要表现在高频部分，MPC 的高频要比 MP3 细腻不少，也没有 OGG 那种金属味道，是目前最适合于音乐欣赏的有损编码。由于都是新生的编码，和 OGG 际遇相似，也缺乏广泛的软件和硬件支持。MPC 有不错的编码效率，编码时间要比 OGG 和 LAME 短不少。

特　点	中高码率下，具有有损编码中最佳的音质表现，高码率下，高频表现极佳
适用于	在节省大量空间的前提下获得最佳音质的音乐欣赏

8）MP3 PRO

作为 MP3 的改良版本的 MP3 PRO 表现出了相当不错的素质，高音丰满，虽然 MP3 PRO 是通过 SBR（Spectral Band Replication，频段复用）技术在播放过程中插入的，但实际听感相当不错，它改善了低位率情况下音频和语音编码的性能。

特　点	低码率下的音质之王
适用于	低要求下的音乐欣赏

9）APE

APE 是 Monkey's Audio 提供的一种无损压缩格式。Monkey's Audio 提供了 Winamp 的插件支持，因此这就意味着压缩后的文件不再是单纯的压缩格式，而是和 MP3 一样的可以播放的音频文件格式。这种格式的压缩比远低于其他格式，能够做到真正无损，因此获得了不少用户的青睐。

特　点	音质非常好
适用于	最高品质的音乐欣赏及收藏

10）AAC 格式

AAC（Advanced Audio Coding，高级音频编码技术）是杜比实验室为音乐社区提供的技术。AAC 号称"最大能容纳 48 通道的音轨，采样率达 96 kHz，并且在 320Kbps 的数据速率

很多听觉不是很好的人，根本就听不到频率为 20000Hz 的声音，不管响度有多大。当人耳同时听到两个不同频率、不同响度的声音时，响度较小的那个也会被忽略。例如，在白天我们很难听到电脑中散热风扇的声音，晚上它却成了噪声源，根据这种原理，编码器可以过滤掉很多听不到的声音，以简化信息复杂度，增加压缩比，而不明显地降低音质。这种遮蔽被称为同时遮蔽效应。每一种频率的临界带宽都不一样，频率越高的临界带宽越宽。如表 1-1 所示为各种频率的临界带宽。

表 1-1　各种频率的临界带宽

频率（Hz）	临界带宽（Hz）	频率（Hz）	临界带宽（Hz）
50	80	1850	280
150	100	2150	320
350	100	2500	380
450	110	3400	550
570	120	4000	700
700	140	4800	900
840	150	5800	1100
1000	160	7000	1300
1170	190	8500	1800
1370	210	10500	2500
1600	240	13500	3500

根据这种效应，专家们设计出了人耳听觉心理模型，这个模型被导入到 MP3 编码中后，掀起了一场翻天覆地的音质革命，MP3 编码技术一直背负着音质差的恶名，但这个恶名现在已经逐渐被洗脱。到了此时，一直被埋没的 VBR 技术现在光彩四射，配合心理模型的运用便显示出强大的诱惑力与杀伤力。

特　点	音质好，压缩比比较高，被大量软件和硬件支持，应用广泛
适用于	比较高要求的音乐欣赏

4）WMA 格式

WMA 就是 Windows Media Audio 编码后的文件格式，它是微软公司推出的与 MP3 格式齐名的一种新的音频格式。WMA 在压缩比和音质方面都超过了 MP3，更是远胜于 RA（Real Audio），即使在较低的采样频率下也能产生较好的音质。WMA 针对的不是单机市场，而是网络！竞争对手就是网络媒体市场中著名的 Real Networks。

特　点	低码率下的音质表现难有对手
适用于	数字电台架设、在线试听、低要求下的音乐欣赏

5）OGG 格式

OGG 全称应该是 OGG Vobis。OGG 并不是一个厂商的名字，而是一个庞大的多媒体开发计划的项目名称，它将涉及视频音频等方面的编码开发。整个 OGG 项目计划的目的就是

下能为 5.1 声道音乐节目提供相当于 ITU-R 广播的品质”。被手机界称为“21 世纪数据压缩方式”。

特　点	支持多个音轨，语言兼容能力强，编码效率高
适用于	音乐网站，音乐社区及音乐节目等

1.1.3　视频的相关基础概念

视频是活动的图像，正如像素是一幅数字图像的最小单元一样，一幅幅静止的图像组成了视频，图像是视频的最小和最基本的单元。视频是由一系列的图像组成的，在电视中把每幅图像称为一帧（Frame），在电影中每幅图像称为一格。

与静止图像不同，视频是活动的图像。当以一定的速率将一幅幅画面投射到屏幕上时，由于人眼的视觉暂留效应，我们的视觉就会产生动态画面的感觉，这就是电影和电视的由来。对于人眼来说，若每秒播放 24 格（电影的播放速率）、25 帧（PAL 制电视的播放速率）或 30 帧（NTSC 制电视的播放速率）就会产生平滑和连续的画面效果。

为了在 Internet 上有效地、高质量地传输视频流，这就需要多种技术的支持，其中数字视频的压缩编/解码技术是 Internet 视频传输中的最为关键的技术之一。

1.1.4　常见的视频文件格式

视频压缩技术和视频编辑处理技术的不断创新和改进使视频文件的体积大幅度减小并且使视频文件的传输性能日益流畅。在平常的工作和学习中，常见的视频文件格式分为普通视频文件格式和网络影像视频格式。

1．普通视频文件格式

1）AVI 格式

AVI 的英文全称为 Audio Video Interleaved，即音频视频交错格式。它于 1992 年由 Microsoft 公司推出，随 Windows 3.1 一起被人们所认识和熟知。所谓“音频视频交错”就是可以将视频和音频交织在一起进行同步播放。这种视频格式的优点是图像质量好，可以跨多个平台使用，其缺点是体积过于庞大，而且更加糟糕的是压缩标准不统一。

2）nAVI 格式

nAVI 是 newAVI 的缩写，是一个名为 Shadow Realm 的地下组织发展起来的一种新视频格式（与我们上面所说的 AVI 格式没有太大联系）。它是由 Microsoft ASF 压缩算法的修改而来的，但是又与下面介绍的网络影像视频中的 ASF 视频格式有所区别，它以牺牲原有 ASF 视频文件视频“流”特性为代价而通过增加帧率来大幅提高 ASF 视频文件的清晰度。

3）DV-AVI 格式

DV 的英文全称是 Digital Video Format，是由索尼、松下、JVC 等多家厂商联合提出的一种家用数字视频格式。目前非常流行的数码摄像机就是使用这种格式记录视频数据的。它

可以通过电脑的IEEE 1394端口传输视频数据到电脑，也可以将电脑中编辑好的视频数据回录到数码摄像机中。这种视频格式的文件扩展名一般是.avi，所以也叫DV-AVI格式。

4）MPEG格式

它的英文全称为Moving Picture Expert Group，即运动图像专家组格式，家里常看的VCD、SVCD、DVD就是这种格式。MPEG文件格式是运动图像压缩算法的国际标准，它采用了有损压缩方法减少运动图像中的冗余信息。目前MPEG格式有3个压缩标准，分别是MPEG-1、MPEG-2和MPEG-4，另外，MPEG-7与MPEG-21仍处在研发阶段。

提示：

细心的读者一定注意到了，这中间怎么没有MPEG-3编码？实际上，大家熟悉的MP3就是采用的MPEG-3（MPEG Layeur3）编码的。

5）DivX格式

这是由MPEG-4衍生出的另一种视频编码（压缩）标准，也即我们通常所说的DVD格式，它采用了MPEG-4的压缩算法同时又综合了MPEG-4与MP3各方面的技术，说白了就是使用DivX压缩技术对DVD盘片的视频图像进行高质量压缩，同时用MP3或AC3对音频进行压缩，然后再将视频与音频合成并加上相应的外挂字幕文件而形成的视频格式。其画质直逼DVD并且体积只有DVD的几分之一。这种编码对机器的要求也不高，所以DivX视频编码技术可以说是一种对DVD威胁最大的新生视频压缩格式。

6）MOV格式

美国Apple公司开发的一种视频格式，默认的播放器是苹果的Quick Time Player。具有较高的压缩比率和较完美的视频清晰度，但是其最大的特点还是跨平台性，即不仅能支持MacOS，同样也能支持Windows系列。

2. 网络影像视频格式

随着互联网和网络技术的发展，网络影像视频格式也变的越来越多，下面介绍几个比较流行的网络影像视频格式。

1）ASF格式

ASF的英文全称为Advanced Streaming Format，它是微软为了和现在的Real Player竞争而推出的一种视频格式，用户可以直接使用Windows自带的Windows Media Player对其进行播放。由于它使用了MPEG-4的压缩算法，所以压缩率和图像的质量都很不错（高压缩率有利于视频流的传输，但图像质量肯定会降低，所以有时候ASF格式的画面质量不如VCD是正常的）。

2）WMV格式

WMV的英文全称为Windows Media Video，也是微软推出的一种采用独立编码方式并且可以直接在网上实时观看视频节目的文件压缩格式。WMV格式的主要优点包括本地或网络回放、可扩充的媒体类型、部件下载、可伸缩的媒体类型、流的优先级化、多语言支持、环境独立性、丰富的流间关系及扩展性等。

3）RM 格式

Real Networks 公司所制定的音频视频压缩规范称为 Real Media，用户可以使用 Real Player 或 Real One Player 对符合 Real Media 技术规范的网络音频/视频资源进行实况转播，并且 Real Media 可以根据不同的网络传输速率制定出不同的压缩比率，从而实现在低速率的网络上进行影像数据的实时传送和播放。这种格式的另一个特点是用户使用 Real Player 或 Real One Player 播放器可以在不下载音频/视频内容的条件下实现在线播放。RM 和 ASF 格式可以说各有千秋，通常 RM 视频更柔和一些，而 ASF 视频则相对清晰一些。

4）RMVB 格式

这是一种由 RM 视频格式升级延伸出的新视频格式，它的先进之处在于打破了原先 RM 格式那种平均压缩采样的方式，在保证平均压缩比的基础上合理利用比特率资源，就是说静止和动作场面少的画面场景采用较低的编码速率，这样可以留出更多的带宽空间，而这些带宽会在出现快速运动的画面场景时被利用。这样在保证了静止画面质量的前提下，大幅地提高了运动图像的画面质量，从而图像质量和文件大小之间就达到了微妙的平衡。

1.2 数字音/视频数据的编/解码技术

数字音/视频的编/解码，也就是要用尽可能少的比特数表示，压缩后传输并将其还原。而压缩后的数字化声音传输具有以下优点。

- 便于集成，便于接口和标准化。
- 易于加密，且保密性强。
- 便于传输，易于纠错，抗信道干扰能力强。
- 压缩后的数据节省了带宽，提供了传输效率。
- 数字化后便于计算机处理和控制，易于与 IP 融合。

因此，可以通过压缩编码技术对声音信息进行高效率的压缩，然后再进行存储和传输。

数字音/视频数据之所以能够压缩是因为基于原始信源的数据存在着很大的冗余度。信息理论认为，若信源编码的熵大于信源的实际熵，则该信源中一定存在冗余度。去掉冗余不会减少信息量，仍可原样恢复数据；但若减少了熵，则数据不能完全恢复。不过，在允许的范围内损失一定的熵，数据仍可以近似地恢复。

衡量一种数据压缩技术的好坏有 3 个重要的指标。

- 压缩比要大。
- 恢复效果要好，要尽可能地恢复原始数据。
- 前两个指标合在一起要求就是，在保证一定观赏质量的前提下尽量提高数据压缩比。实现压缩的算法要简单，压缩、解压缩速度快，尽可能做到实时压缩、解压缩。

随着数字通信的发展和计算机技术的发展，数据压缩技术已日臻成熟，适合各种应用场合的编码方法不断产生。目前常用的压缩编码方法分为两大类：一类是无损压缩法，也称为冗余度压缩法或熵编码法；另一类是有损压缩法，也称熵压缩法。

目前已发展和正在发展的数字视频和音频压缩技术有很多种，不同的压缩技术有不同的侧重点，适应不同的应用。这些压缩技术中有的已经标准化，但还有很多并没有标准化。常用的已经标准化的压缩技术有 MPEG-1、MPEG-2、H.261/H.263 等，正在发展的有 MPEG-4 等。MPEG-1、MPEG-2 适用于高带宽的能够提供高质量低延迟的视频和音频应用，而 H.261、H.263 及正在发展的 MPEG-4 则适用于低带宽的对图像质量的延迟要求不高的应用。

数字音/视频数据的编/解码通常包括音频数据的编/解码和视频数据的编/解码。

1.2.1 音频数据的编/解码

音频数据的编/解码实际上由两个部分组成。第一部分称为编码，把存放在波形文件里的数字音频数据转换为高度压缩的形式，称为比特流（或者叫已编码音频数据）。要在声卡上播放比特流时，需要第二部分，即解码。解码把比特流重建为波形文件。

在音频压缩出现之前，存储高质量的数字音频数据往往需要占用大量的硬盘空间（或者通道带宽）。下面让我们来看一个简单的例子。如果要对喜欢的歌曲进行采用并存储，假设歌曲是一分钟，要想达到 CD 音质，就要以 44.1kHz 的采用频率、立体声和每个采样值 16bit 的方式进行采样。44.1kHz 意味着每秒要有 44 100 个采用值从声卡里输出。因为有 2 个通道，所以乘以 2。又因为每个采样值有 2 个字节（即 16bit），所以再乘以 2。那么总的数据量是 44 100（采样值/秒）×2（2 个通道）×2（2 个字节/采样值）×60（秒）=10 584 000（字节），故大约 10MB 的硬盘空间。如果想从 Internet 上下载这首歌的话，使用一个普通的 28.8Kbit/s 的 Modem，如果不算传输开支和网络不畅，理论上将会用 1 058 000（字节）×8（8 位/字节）/28 000（位/秒）×60（秒/米）=49 分钟，这样才仅仅能下载一分钟的歌曲。这对大部分用户是不可接受的，尤其是喜欢在电脑上听音乐的朋友，要降低磁盘占用空间，只有两种方法，降低采样指标或者压缩。降低指标是不可取的，因此必须要研发各种压缩方法。

音频压缩的目的就是利用音频数据中存在大量的冗余度，来减少音频数据所需的存储空间或网络通道带宽。一般获取音频数据的方法是：采用固定的时间间隔对音频电压采样（量化），并将结果以某种分辨率（例如：CDDA 每个采样为 16 比特或 2 字节）存储。采样的时间间隔可以有不同的标准，如 CDDA 采用每秒 44 100 次；DVD 采用每秒 48 000 或 96 000 次。因此，采样率、分辨率和声道数目（例如立体声为 2 声道）是音频文件格式的关键参数。

同时需要分清楚的是音频文件和编/解码器不同。尽管一种音频文件格式可以支持多种编码，例如 AVI 文件格式，但多数的音频文件仅支持一种音频编码。有两类主要的音频文件格式。

- 无损格式：例如 WAV、PCM、APE、TTA、FLAC、AU 等。
- 有损格式：例如 MP3、OGG Vobis（OGG）、Windows Media Audio（WMA）、AAC 等。

1．无损压缩

所谓无损压缩音频格式，顾名思义，就是毫无损失地将声音信号进行压缩的音频格式。常见的像 MP3、WMA 等格式都是有损压缩格式，相对于作为源的 WAV 文件，它们都有相当大程度的信号丢失，这也是它们能达到 10%的压缩率的根本原因。而无损压缩格式，就好比用 Zip 或 RAR 这样的压缩软件去压缩音频信号，得到的压缩格式还原成 WAV 文件，和作为源的 WAV 文件是一模一样的！但是如果用 Zip 或 RAR 来压缩 WAV 文件的话，必须将压

缩包解压后才能播放。而无损压缩格式则能直接通过播放软件实现实时播放，使用起来和MP3等有损格式一模一样。总而言之，无损压缩格式就是能在不牺牲任何音频信号的前提下减少WAV文件体积的格式。

当然，我们这里所说的无损压缩，和自然、真实的声音相比还是有损的——作为数字音频格式，只能做到无限接近于无损，想要完全做到无损显然是不可能的。

相对于有损压缩的一些格式，无损压缩格式对很多玩家来说还比较陌生，事实上无损压缩的技术已经很成熟了，下面将介绍其中几种比较常见的格式。

- FLAC：压缩率在4种格式中最差，编码速度不错，平台支持很好。
- PAC：稍慢的编码速度，压缩率排第三，平台支持良好。
- APE：编码速度最快、最好的压缩率，平台支持一般。
- WV：编码速度非常快，压缩率在4种格式中排第二，仅支持Windows平台。

2. 有损压缩

有损压缩方法也称信息量压缩，这种压缩方法利用了人类视觉或者人类听觉对图像或声音中的某些频率成分不敏感的特性，从原始数据中将这一部分人类视觉或者人类听觉不敏感的数据去除，以达到压缩的目的。不能完全恢复原始数据是有损压缩方法的最大缺点，但是所损失的部分对理解原始图像或者倾听原始声音的影响较小，却换来了大得多的压缩比，因此，有损压缩广泛应用于语音、图像和视频数据的压缩。

有损压缩应用广泛，但在专业领域使用不多。有损压缩具有很大的压缩比，提供相对不错的声音质量。例如下面几个例子。

- AAC：在高比特率下音质仅次于 MPC，在高比特率和低比特率下表象都很不错。就是编码速度太慢。
- MPC：在低比特率下表现一般，不及MP3Pro编码的MP3和OGG，在高比特率下音质最好，编码速度快。
- OGG：在低比特率下音质最好，在高比特率下也不错，编码速度稍慢。
- MP3（MP3Pro）：在低比特率下音质次于OGG，在其他方面同MP3一样。
- WMA：在高低比特率下都一般，不支持VBR，最高192Kbit/s。

下面是各种有损压缩编码格式的相互比较，如表1-2所示。

表1-2　各种有损压缩编码格式比较

编码方式	低比特率品质	高比特率品质	压缩率	编码速度
ACC	一般	好	高	慢
MP3	好	好	高	快
MPC	一般	最好	高	快
OGG	最好	一般	高	慢
WMA	一般	一般	高	快

1.2.2 视频数据的编/解码

视频压缩（编/解码）的目标是在尽可能保证视觉效果的前提下降低视频数据率。视频压缩比一般指压缩后的数据量与压缩前的数据量之比。由于视频是连续的静态图像，因此其压缩编码算法与静态图像的压缩编码算法有某些共同之处，但是运动的视频还有其自身的特性，因此在压缩时还应考虑其运动特性才能达到高压缩的目标。在视频压缩中常需用到以下一些基本概念。

1．有损和无损压缩

在视频压缩中有损（Lossy）和无损（Lossless）的概念与静态图像中基本类似。无损压缩也即压缩前和解压缩后的数据完全一致。多数的无损压缩都采用 RLE 行程编码算法。有损压缩意味着解压缩后的数据与压缩前的数据不一致。在压缩的过程中要丢失一些人眼和人耳所不敏感的图像或音频信息，而且丢失的信息不可恢复。几乎所有高压缩的算法都采用有损压缩，这样才能达到低数据率的目标。

2．帧内和帧间压缩

帧内（Intraframe）压缩也称为空间压缩（Spatial Compression）。当压缩一帧图像时，仅考虑本帧的数据而不考虑相邻帧之间的冗余信息，这实际上与静态图像压缩类似。帧内一般采用有损压缩算法，由于帧内压缩时各个帧之间没有相互关系，所以压缩后的视频数据仍可以以帧为单位进行编辑。帧内压缩一般达不到很高的压缩。

帧间（Interframe）压缩也称为时间压缩（Temporal Compression），它通过比较时间轴上不同帧之间的数据进行压缩。帧间压缩一般是无损的。帧差值（Frame Differencing）算法是一种典型的时间压缩法，它通过比较本帧与相邻帧之间的差异，仅记录本帧与其相邻帧的差值，这样可以大大减少数据量。

3．对称和不对称编码

对称性（Symmetric）是压缩编码的一个关键特征。对称意味着压缩和解压缩占用相同的计算处理能力和时间，对称算法适合于实时压缩和传送视频，如视频会议应用就以采用对称的压缩编码算法为好。而在电子出版和其他多媒体应用中，一般是把视频预先压缩处理好，尔后再播放，因此可以采用不对称（Asymmetric）编码。不对称或非对称意味着压缩时需要花费大量的处理能力和时间，而解压缩时则能较好地实时回放，也即以不同的速度进行压缩和解压缩。

1.2.3 音/视频编码的相关标准

国际上音/视频编/解码标准主要有 2 大系列：ISO/IEC JTC1 制定的 MPEG 系列标准；ITU 针对多媒体通信制定的 H.26x 系列视频编码标准和 G.7 系列音频编码标准。经过十年多的演变，音/视频编码技术本身和产业应用背景都发生了明显变化。目前音/视频产业可以选择的信源编

码标准有 4 个：MPEG-2、MPEG-4、MPEG-4 AVC（简称 AVC，也称 JVT、H.264）和 AVS。

1．国际电联的 H.261～H.264 标准

- H.261：H.261 又称为 P*64，其中 P 为 64Kbps 的取值范围，是 1～30 的可变参数，它最初是针对在 ISDN 上实现电信会议应用特别是面对面的可视电话和视频会议而设计的。实际的编码算法类似于 MPEG 算法，但不能与后者兼容。H.261 是最早的运动图像压缩标准，它详细制定了视频编码的各个部分，包括运动补偿的帧间预测、DCT 变换、量化、熵编码，以及与固定速率的信道相适配的速率控制等部分。
- H.262（又称 MPEG-2）：由 MPEG-1 扩充而来，支持隔行扫描。使用十分广泛，几乎用于所有的数字电视系统，适合标清和高清电视，适合各种媒体传输，包括卫星、有线、地面等，都能有效地传输。
- H.263：H.263 视频压缩编码标准是由国际电信联盟制定的用于在低码率条件下实现视频图像服务的压缩技术。但实际上这个标准可用在很宽的码流范围，而非只用于低码流应用，它在许多应用中可以被认为用于取代 H.261。H.263 的编码算法与 H.261 一样，但做了一些改善和改变，以提高性能和纠错能力。

H.263 标准是在早先的 H.261 标准上提出的，并继承了 MPEG-1 和 MPEG-2 标准的优点，它可以提高宽带利用率，以 128Kbit/s 的速率实现全运动视频（30fps）；也可以 20Kbit/s～25Kbit/s 的带宽传输视频流。

- H.264：H.264 是由 ISO/IEC 与 ITU-T 组成的联合视频组（JVT）制定的新一代视频压缩编码标准。H.26L 标准与以前的任何标准相比，效率要高出一倍，同时具有简单、直观的视频编码技术，网络友好的视频描述，适合交互和非交互式应用（广播、存储、流媒体）。

2．MPEG 系列标准

MPEG 是活动图像专家组（Moving Picture Exports Group）的缩写，于 1988 年成立，是为数字视/音频制定压缩标准的专家组，目前已提出 MPEG-1、MPEG-2、MPEG-4、MPEG-7 和 MPEG-21 标准。

- MPEG-1 标准：MPEG-1 标准于 1993 年 8 月公布，用于传输 1.5Mbps 数据传输率的数字存储媒体运动图像及其伴音的编码。该标准包括 5 个部分：第 1 部分说明了如何根据第 2 部分（视频）及第 3 部分（音频）的规定对音频和视频进行复合编码。第 4 部分说明了检验解码器或编码器的输出比特流符合前 3 部分规定的过程。第 5 部分是一个用完整的 C 语言实现的编码和解码器。
- MPEG-2 标准：MPEG-2 标准是针对标准数字电视和高清晰度电视在各种应用下的压缩方案和系统层的详细规定，编码码率为每秒 3 兆比特～每秒 100 兆比特，标准的正式规范在 ISO/IEC13818 中。MPEG-2 不是 MPEG-1 的简单升级，MPEG-2 在系统和传送方面做了更加详细的规定和进一步的完善。MPEG-2 特别适用于广播级的数字电视的编码和传送，被认定为 SDTV 和 HDTV 的编码标准。
- MPEG-4 标准：MPEG-4 与 MPEG-1 和 MPEG-2 有很大的不同。MPEG-4 不只是具体压缩算法，它是针对数字电视、交互式绘图应用（影音合成内容）、交互式多媒体（WWW、资料撷取与分散）等整合及压缩技术的需求而制定的国际标准。MPEG-4

标准将众多的多媒体应用集成于一个完整的框架内，旨在为多媒体通信及应用环境提供标准的算法及工具，从而建立起一种能被多媒体传输、存储、检索等应用领域普遍采用的统一数据格式。

- MPEG-7 标准：MPEG-7 标准被称为“多媒体内容描述接口”，为各类多媒体信息提供一种标准化的描述，这种描述将与内容本身有关，允许快速和有效地查询用户感兴趣的资料。它将扩展现有内容识别专用解决方案的有限的能力，特别是它还包括了更多的数据类型。换言之，MPEG-7 规定一个用于描述各种不同类型多媒体信息的描述符的标准集合。

3. MPEG-21 标准

MPEG-21 标准其实就是一些关键技术的集成，通过这种集成环境就对全球数字媒体资源进行透明和增强管理，实现内容描述、创建、发布、使用、识别、收费管理、产权保护、用户隐私权保护、终端和网络资源抽取、事件报告等功能。

4. 其他压缩编码标准

- Real Video：Real Video 是 Real Networks 公司开发的在窄带（主要的互联网）上进行多媒体传输的压缩技术。
- WMT：WMT 是微软公司开发的在互联网上进行媒体传输的视频和音频编码压缩技术，该技术已与 WMT 服务器与客户机体系结构结合为一个整体，使用 MPEG-4 标准的一些原理。
- Quick Time：Quick Time 是一种存储、传输和播放多媒体文件的文件格式和传输体系结构，所存储和传输的多媒体通过多重压缩模式压缩而成，传输是通过 RTP 协议实现的。

1.3 音/视频处理平台的选择

数字视频可以采用各种的形式与格式，开发人员需要支持繁杂的配置和各种不同的方面，如不同的分辨率/显示器尺寸，不同的比特率、实时问题甚至视频源的可靠性等（例如，来自硬盘驱动器的视频流与来自无线通信链路的视频流的区别）。即使是那些看似简单的任务，如高效管理音频/视频同步及在 IP 网络上实现可靠的视频传输，也会让开发人员伤透脑筋。虽然有些器件让众多应用开发人员大喜过望，但是它们不得不支持多种标准，其中包括那些必须着手制定的新标准。下面简单介绍一下常见的几个音/视频开发平台。

1.3.1 个人多媒体计算机

在多媒体计算机之前，传统的微机或个人机处理的信息往往仅限于文字和数字，只能算是计算机应用的初级阶段，同时，由于人机之间的交互只能通过键盘和显示器来实现，故交流信息的途径缺乏多样性。为了改换人机交互的接口，使计算机能够集声、文、图像处理于一体，人类发明了有多媒体处理能力的计算机。

多媒体计算机是能够对声音、图像、视频等多媒体信息进行综合处理的计算机。多媒体计算机一般指多媒体个人计算机（Multimedia Personal Computer，MPC），1985 年出现了第一台多媒体计算机，其主要功能是把音频视频、图形图像和计算机交互式控制结合起来，进行综合处理。与计算机系统类似，多媒体计算机系统由多媒体硬件和多媒体软件构成。

1.3.2 嵌入式处理器ARM

ARM7 系列广泛应用于多媒体和嵌入式设备，包括 Internet 设备、网络和调制解调器设备，以及移动电话、PDA 等无线设备。无线信息设备领域的前景广阔，因此，ARM7 系列也瞄准了下一代智能化多媒体无线设备领域的应用；ARM9 系列主要应用于引擎管理、仪器仪表、安全系统、机顶盒、高端打印机、PDA、网络电脑及带有 MP3 音频和 MPEG4 视频多媒体格式的智能电话中；ARM10 系列极大地提高了处理器的整型和浮点运算性能，为用户界面的 2D 和 3D 图形引擎应用夯实基础，如视频游戏机和高性能打印机等。

随着多媒体技术的发展，视频、音频信息的处理水平越来越高，为嵌入式系统的多媒体化创造了良好的条件。后 PC 时代出现的各种信息电器，如掌上电脑、可视电话、数码相机等嵌入式设备，表明嵌入式系统的多媒体化正逐步变成现实。

1.3.3 数字媒体处理器DSP

数字信号处理器，也称 DSP 芯片，是针对数字信号处理需要而设计的一种具有特殊结构的微处理器，它是现代电子技术、计算机技术和信息处理技术相结合的产物。由于数字信号处理器运算速度快，具有可编程和接口灵活的特点，使得它在各个方面、各个领域的应用越来越广泛。特别是在音/视频领域得到了飞速发展。

随着技术的飞速发展，DSP 在数字 GSM 手机应用和无线基站应用中获得了巨大的成功。目前 DSP 开始全面扩展到应用领域，在宽带通信、数字控制、数字音频、数字视频等市场上开始占据主角的地位。

1.4 音/视频开发的典型应用

随着多媒体技术的发展和人们生活水平的提高，音/视频技术在日常工作和生活中的应用越来越多，下面简要介绍一下音/视频的几个开发应用。

1.4.1 个人消费类电子产品

随着数码影音领域的持续发展，大众对便携式娱乐产品的需求不断增加，越来越强大的记忆存储设备和网上影音内容的出现拓展了便携式娱乐设备的领域。目前，便携式个人消费类播放器产品有 CD 播放器、MP3 音频播放器和便携式多媒体播放器（PMP）等 3 大类。这 3 类产品特点各异，价格也是由低到高，各自有具明确的用户群和价位。如图 1-2 所示为常

见的个人消费类电子产品。

（a）CD 播放器　　（b）MP3 播放器　　（c）MP4 播放器

图 1-2　个人消费类电子产品

1. CD 播放器

CD 机实际上就是我们在日常生活中常说的 CD 播放器（如图 1-2（a）所示），也有人把它称为激光唱机或是镭射唱机。它是一种用微电脑控制的智能化高保真立体声音响设备，采用了先进的激光技术、数码技术、计算机技术和各种新型元器件，具有高密度记录、放音时间长、操作简便、选曲快速等优点。它能逼真地重放录制的内容，层次分明，有临场感。其音响技术指标很高，动态范围大，频响宽度为 5Hz～2000Hz，失真度小到 0.003%。

目前 CD 播放器已经是非常普遍的音响播放设备了，根据它的外形可以将具分为台式和便携式两种。最出名的便携式 CD 机有索尼、松下等。

2. MP3 数字音频播放器

随着微电子技术和数字电子技术的发展，为满足市场的需求，基于 MPEG Layer-3 的 MP3 数字音频播放器（如图 1-2（b）所示）作为独立的新产品应运而生，迅速地被广大的消费者接受。

MP3 音乐播放主要是通过类似“Win Amp”的播放软件进行的，在一定程度上方便了广大的计算机使用者和音乐爱好者，但这种不可脱离 PC 的音乐播放方式也为人们欣赏音乐带来了不便。于是，美国 Diamond 公司和韩国的 Saehan 公司率先推出了 MP3 随身听，正式启动了 MP3 数字音频播放器（以下简称 MP3）的市场。

MP3 数字音频播放器通常分为闪存式 MP3 和硬盘式 MP3。目前一些电子产品也嵌入了 MP3 功能，如手机、PDA、车载音响和数码相机等。MP3 是采用数字电子技术构建的录放音机，它还可作为数字电子存储器（U 盘）使用，多数 MP3 还集成了调频立体声收音功能，因此很受消费者的欢迎。

3. 便携式多媒体播放器（PMP）

随着大众对便携式娱乐产品的需求不断增加，便携式多媒体播放器（Portable Multimedia Playe，也称 MP4，即 MPEG-4 播放器（如图 1-2（c）所示）已逐渐成为数码影音领域中便携式娱乐设备的主流。

音/视频是 PMP 的核心功能，而在此基础之上，PMP 可以扩展诸如视频录制、摄像/照相、数码相册、收音机、电子书、游戏、上网等丰富的功能，成为一种个人信息和娱乐的便携式设备。

MP4 与 MP3 播放器的最大不同就在于可存储与播放数码相机内的照片和个人电脑里的

影片。当然影片文件大小动辄几 GB，所以目前的 MP4 播放器大多选择容量较大的硬盘作为存储媒介（当然记忆卡也是必不可少的），因此 PMP 的体积相比于轻灵小巧的 MP3 播放器就显得比较臃肿了。大部分的 MP4 播放器都拥有优异的 TFT-LCD 显示屏，不论是相片还是影片的品质都十分细腻。有的 PMP 电池持航能力更是可以连续播放长达 5 小时，足以看完两部电影。同时，在 PMP 平台上，可支持视听娱乐、移动办公、视频上网、GPS 卫星导航，它是一款功能全面，性能超群的“个人数字多媒体平台”，对于追求声色娱乐及时尚品味的人来说，它既是炫酷的玩物，更是得力的商务助理。

PMP 未来的技术发展趋势包括操作系统更多地倾向于采用 Linux；为了在家里或宾馆里用大屏幕电视来观看 PMP 上的视频文件，PMP 上需要增加 TV 输出端子；为了更容易地从 PC 或 DV 下载多媒体内容，越来越多的客户选择在 PMP 上实现 USB 2.0、USB OTG 和 IEEE1394 接口；为了在出差途中或休闲旅游路上接收并播放流式多媒体内容，PMP 开始融合数字电视和数字卫星/地面/网络收音机等功能。此外，针对中国越来越庞大的汽车高消费群体的车载多媒体播放机也开始进入系统开发商的视野。

1.4.2　公共领域的音/视频应用

目前，多媒体技术和网络技术已经应用到很多公共领域，如语音识别、视频会议和远程监控（如图 1-3 所示）等，正是由于音/视频的不断发展和应用，逐渐改变了人们的工作和生活方式。下面介绍一下音/视频技术在公共领域的几个应用。

（a）语音识别系统　　（b）视频会议系统　　（c）远程视频监控系统

图 1-3　公共领域音/视频应用

1．语音识别系统

语音识别技术就是让电脑听懂人的话，并做出正确反应。计算机对语音识别处理过程与人对语音识别处理过程基本上是一致的。目前主流的语音识别技术是基于统计模式识别的基本理论。一个完整的语音识别系统（如图 1-3（a）所示）可分为 3 部分。

- 语音特征提取：目的就是从语音波形中提取初随时间变化的语音特征序列。
- 声学模型与模式匹配（识别算法）：声学模型通常将获取的语音特征通过学习算法产生。在识别时输入的语音特征与声学模型进行匹配和比较，得到最佳的识别结果。
- 语言模型与语言处理：语言模型包括由识别语音的命令构成的语法网络，或由统计方法构成的语言模型，语言处理可以进行语法、语义分析。

声学模型是识别系统的底层模型，并且是语音识别系统中最关键的一部分。建立声学模型的目的是提供一种有效的方法计算语音的特征矢量序列和每个发音模板之间的距离。

现在关于语音识别研究重点集中于下面几个方面。

- 电子发音：让计算机来读文章，这是人与计算机沟通的一个重要方面，这方面英文电子发音研究成果很多，中文处于起步阶段。
- 语音控制：用人声控制计算机的动作。
- 语音训练：这是语音识别的一个较新的方向，它要求人模仿标准发音，面临的困难是如何衡量人模仿的好坏。
- 连续语音识别：中英文连续语音识别技术都已进入使用阶段，对于中文、多音字的识别有了很大进展，但依旧存在对多音字识别的问题，目前主要的识别错误基本都来自此类问题。
- 非连续语音识别：对有限词的识别技术相对完善，但大词汇量非特定人等方面的识别效果很差。

语音识别技术的应用可以分为两个发展方向：一个方向是大词汇量连续语音识别系统，主要应用于计算机的听写机，以及与电话网或者互联网相结合的语音信息查询服务系统，这些系统都是在计算机平台上实现的；另外一个重要的发展方向是小型化、便携式语音产品的应用，如无线手机上的拨号、汽车设备的语音控制、智能玩具、家电遥控等方面的应用，这些应用系统大都使用专门的硬件系统实现，特别是近几年来迅速发展的语音信号处理专用芯片（Application Specific Integrated Circuit，ASIC）和语音识别片上系统（System on Chip，SOC）的出现，为其广泛应用创造了极为有利的条件。

2．视频会议系统

视频会议系统（Video Conference）（如图1-3（b）所示）是指两个或两个以上不同地方的个人或群体通过传输线路及多媒体设备，将声音、影像及文件资料互传，进行即时且互动的沟通。

视频会议系统是集通信、计算机技术、多媒体技术于一体的远程异地通信方式。在召开会议时，处于两地或多个不同地点的与会代表，既可以听到对方的声音，又能看到对方的形象，同时还能看到对方会议室的场景，以及会议中展示的实物、图片、表格、文件等，与真实的会议没什么区别，使每个与会者都有身临其境之感。在实际应用中，视频会议能够提高工作效率，降低远距离会议的费用。根据运行环境和支持标准的不同可以将视频会议系统划分为以下几个不同的类型。

- 根据会议节点数目的不同，可以将视频会议系统分为点对点视频会议系统和多点视频会议系统。点对点视频会议系统应用于两个通信节点之间。多点视频会议系统应用于两个以上的节点之间的通信。
- 根据运行的通信网络不同，可以将视频会议系统分为数字数据网（DDN）或其他专用网型、局域网/广域网（LAN/WAN）和公共交换电话网型（PSTN）3种。使用DDN或专业网，在（384～2048）Kbit/s速率下，可提供（25～30）帧/s的CIF或QCIF图像；在LAN/WAN环境中，运行在384Kbit/s速率下，可提供（15～20）帧/s的图像；而在PSTN上，运行在28.8Kbit/s或33.6Kbit/s等速率下，只能达到（5～10）帧/s。
- 根据技术支持的类型的不同，视频会议系统可分为基于线路的视频会议系统和基于分组的视频会议系统，但是现在两者间的界限越来越模糊。基于线路的视频会议系

统也称为常规视频编/解码系统，依照专用线路提供一个确定的比特率，诸如租用线路或公用线路交换服务。基于分组的视频会议系统是从分组视频通话系统演化而来的，其基本原理是相同的，即利用计算机支持视频会议的服务。

- 根据所选用的终端类型不同，视频会议系统又可分为桌面视频会议系统（Desktop Video Conference）、会议室型视频会议系统（Room/Roll-about Video Conference）和可视电话系统。

桌面型视频会议系统是在普通计算机上增加一些附加设备，主要利用计算机软件完成会议功能。桌面视频会议日益受到人们的青睐，因为它有效地利用了现有的资源。

会议室视频会议系统在带有环境控制设备的专用会议房间里装置一个或多个大屏幕，系统由屏幕、摄像机、麦克风和辅助设备组成。这些配置是永久性的，它们不能移到别的房间或大楼内，但可以提供高质量的视频和同步音频。

可视电话系统用于点到点通信，它满足了在电话上进行视频会议传输的需求。系统组成包括一个小屏幕、内部摄像机、音频系统、视频编辑器和键盘。

典型的多媒体视频会议系统由终端设备、通信链路、多点控制单元（Multi-point Control Unit，MCU）及相应的软件部分组成。

视频会议系统的音频编/解码技术：H.323 框架协议下的 G 系列标准协议 G.711、G.722、G.723、G.728 等，是 ITU 为语言通信推荐的系列协议标准，音频占用 64kHz 以下的带宽，音质相当于电话效果（G.722 能达到调频广播的音质）。

3. 远程视频监控系统

视频监控系统（如图 1-3（c）所示）对于保障人们日常生产和生活的安全具有很重要的意义，是大型企业如集团公司、邮电、银行等信息交流广泛的企业生产与管理的必备系统。视频监控系统经历了 3 个发展时代。

远程视频监控系统是以计算机技术为核心，结合先进的多媒体技术、网络通信技术、数字图像压缩技术的一种远程监控系统。远程监控系统能将监控现场的监控信息通过计算机网络传输到网络中的其他计算机上，并与信息管理系统融合在一起，达到远程监控的目的。

一个比较典型的远程监控系统，它涉及远程监控现场构成、监控中心构成、视频进入计算机网络、视频图像传输、现场及远程多点控制等多方面的内容。一般远程监控系统包括本地和远程监控两部分，由远程监控现场和监控中心组成，在远程监控现场和监控中心之间通过网络通信线路链接，其结构如图 1-4 所示。

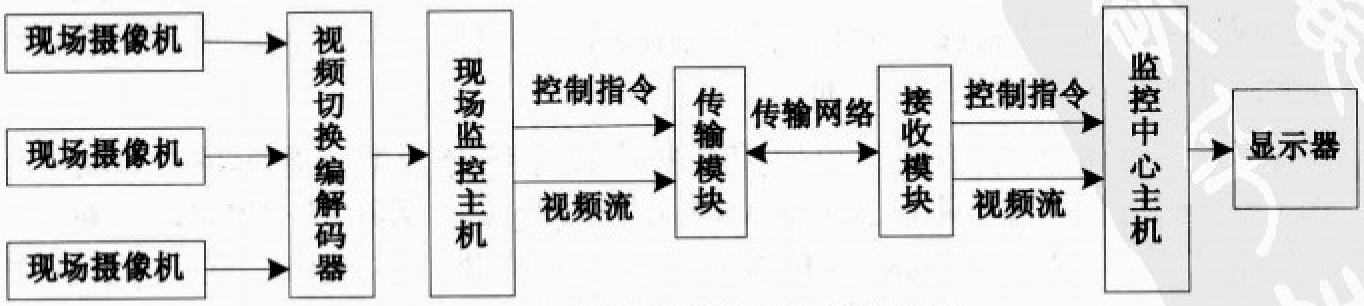


图 1-4　远程视频监控系统的结构图

每个远程监控现场均有若干摄像机。摄像机的镜头、云台可控，并加装传感器、警灯的外围报警设备。每个监控现场由一台配置相对较好的计算机作为基本监控中枢，内含视频采集卡和视频处理软件，它对监控现场的视频录像进行实时播放、采集及存储，同时还要向监

控中心发送实时图像。还可以控制现场的摄像机切换，镜头、云台动作及处理警报信息等。同时它也能接受监控中心发送的控制命令，并执行相应的操作。

监控中心由一台或多台配置相对较高的监控主机组成。在监控中心可以任意对各个现场实行实时监控。它既可以完成对远程现场视频图像数据的实时接收、播放、存储及控制该现场的摄像机切换镜头、云台动作，又可以接收处理各现场的报警信息。监控人员将在监控中心完成对远程监控现场的监控工作，以及对设备报警进行处理。

主机、视频采集卡、视频处理软件、图像发送及主控软件，构成了远程监控现场中心单元，摄像机及音频、报警采集器为前端采集单元，网络工作站及监控中心接收控制软件为用户接收单元，各部分协调工作，形成有机而完整的现代化远程视频监控系统。

1.5 常用音/视频开发工具

随着网络的流行和多媒体技术的发展，音/视频开发工具显得日益重要。微软推出的 DirectX 中的 DirectSound、DirectShow 等技术成为了主要的音/视频开发工具。由于 DSP 芯片在音/视频开发中所展现出来的优越性，则用 DSP CCS 算法进行音/视频开发也是不错的选择。

1.5.1 DirectSound音频开发

DirectSound 是 DirectX API 的音频组件之一，它可以提供快速的混音、硬件加速功能，并且可以直接访问相关设备，当然，最主要的是它提供的功能与现有的设备驱动程序保持兼容性。

DirectSound 是 DirectX Audio 的一个较底层的部件，提供了丰富的接口函数，可以实现 WAV 格式的波形声音数据的播放控制。与一般的 Windows API 提供的声音播放函数不同，DirectSound 可实现多个声音的混合播放，也可充分使用声卡的内存资源，同时也提供了 3D 生效算法，模拟出真实的 3D 立体声。

DirectSound 允许进行波形声音的捕获、重放，也可以通过控制硬件和相应的驱动来获得更多的服务。DirectSound 的优势当然和 DirectX 的其他组件一样——速度，它允许最大效率地使用硬件，并拥有良好的兼容性。DirectSound 在音频方面的使用和开发会在以后章节详细说明。

1.5.2 DirectShow视频开发

DirectShow 是个功能强大的视频开发工具，它为多媒体数据流的捕获和回放提供了强有力的支持。运用 DirectShow，我们可以很方便地从支持 WDM 驱动模型的采集卡上捕获数据，并且进行相应的后期处理甚至存储到文件中它广泛支持的各种媒体格式，包括 WAV、AVI、MPEG、MP3、ASF 和 DV 等，使得多媒体数据的回放轻而易举。此外，DirectShow 还集成了 DirectX 其他部分（如 DirectDraw、DirectSound）的技术，直接支持 DVD 的播放、视频的非线性编辑，以及数字摄像机的数据交换。DirectShow 提供的是一种开放式的开发环境，可

以根据自己的需要定制自己的组件，DirectShow 的系统框架如图 1-5 所示。

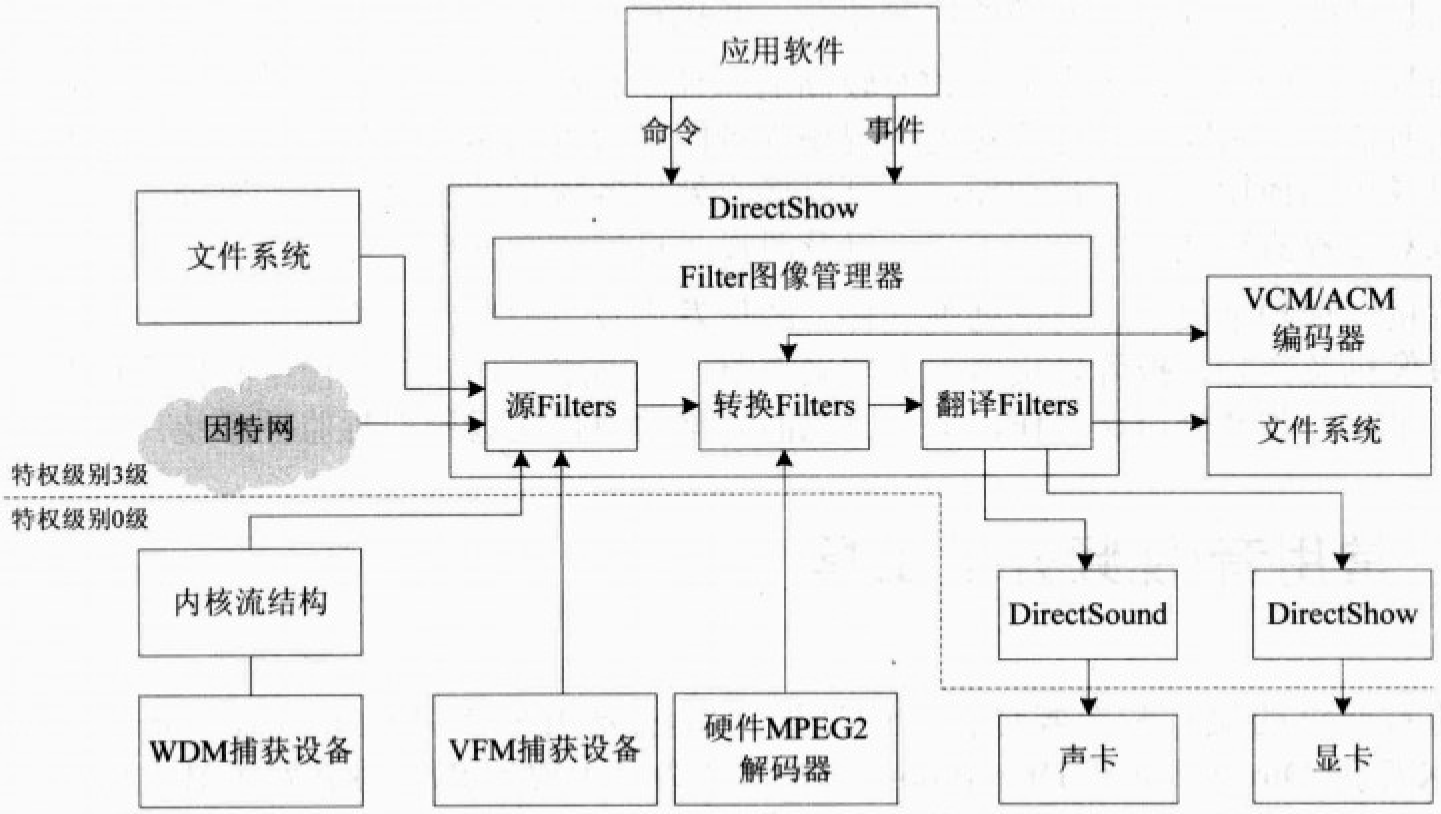


图 1-5　DirectShow 系统框架图

从图 1-5 中可以看到，DirectShow 的滤波器支持多种外部设备，包括本地文件系统、视频采集卡、有线电视接口、视频播放卡（通过 DirectDraw 或 GraphicDevice Interface）、音频播放卡（通过 DirectSound）。虽然 DirectShow 基于 COM 技术，但是如果只是用 DirectShow 开发自己的应用程序而不涉及开发自己的 DirectShow 组件，就不需要了解太多的 COM 的知识。尽管 Microsoft 提供的标准 DirectShow 开发包已经支持了很多格式，但毕竟不是囊括一切的，如它就不支持解码。但是由于是一个开放系统，它允许用户根据自己的需要按照一定规范编写自己的滤波器（也就是 COM），扩充 DirectShow 的功能。

在 DirectShow 系统上，应用程序要按照一定的意图建立起相应的 Filter Graph，然后通过 Filter Graph Manager 来控制整个的数据处理过程。DirectShow 能在 Filter Graph 运行的时候接收到各种事件，并通过消息的方式发送到应用程序。这样，就可以实现应用程序与 DirectShow 系统之间的交互。

1.5.3　DSP CCS算法开发

由 TI 公司发布的 DSP 集成开发环境 CCS（Code Composer Studio），是一个基于 Windows 的 DSP 开发平台，是目前最优秀、最流行的 DSP 开发软件之一。CCS 集成可视化的编辑界面可直接编写 C 语言、汇编语言，除了扩展了基本的代码产生工具外，CCS 还集成了 C 编译器，C 优化器、汇编器、汇编优化器和链接器等，并支持 RTDX（Real Time Data Exchange）技术，可在不中断目标系统运行的情况下，实现 DSP 与其他应用程序（OLE）的数据交换。此外，CCS 的断点工具、探针工具和分析工具使得开发者的一切开发过程都是在 CCS 这个集成环境下进行的，包括项目的建立、源程序的编辑及程序的编译和调试。

CCS 有两种工作模式，即软件仿真器和硬件在线编程。软件仿真器工作模式可以脱离 DSP 芯片，在 PC 上模拟 DSP 的指令集和工作机制，主要用于前期算法实现和调试。硬件在

线编程可以实时运行在 DSP 芯片上，与硬件开发板相结合进行在线编程和调试应用程序。

CCS 的这些优点可以加速开发进程、缩短开发周期、提高工作效率。

1.6 本章小结

本章讲述了数字音/视频开发的基本概念，然后给出了常见的一些音频和视频的格式。由于数字音/视频的编/解码技术在 Internet 及各方面得到广泛的应用，于是详细介绍了数字音/视频编/解码的技术和标准。随着数字音/视频技术的飞速发展，音/视频开发平台也得到了拓展，随后介绍了目前几种常见的音/视频开发平台。同时也介绍了音/视频技术在个人和公共领域的几个应用。

PDG

第 2 章

Visual C++ 2005 开发基础

Visual C++ 2005 是 Microsoft 公司开发的基于 C/C++的集成开发工具，是一种公认的功能强大的、效率高的开发工具。Visual C++ 2005 在原有的基础上增加了新特性，不仅支持原来的 ISO/ANSI 标准 C++，而且还支持名为 C++/CLI（Common Language Infrastructure，CLI）的新版 C++。

本章重点

- Visual C++ 2005 新增特性
- VC 能做的事情
- 利用向导完成基本开发
- MFC 编程基础
- 基于对话框的编程
- 常用控件使用技巧
- Visual 2005 开发技术资源
- Visual 2005 开发常见问题

2.1 Visual C++ 2005新增特性

Visual C++ 2005 提供了强大的新语法支持。它使用的新技术已经使 Microsoft 产品的运行速度提高了 30%，还通过新的编译模式来确保 Microsoft.NET 框架通用语言基础结构（CLI）的一致性和可验证性，Visual C++ 2005 新增语法主要有下列几项。

2.1.1 句柄（Handles）

在旧的语法中，很容易混淆的地方是指针和引用都使用“*”符号，而在 C++/CLI 中引入了句柄的概念，例如：

```
void_tmain()
{
   String^str="Hello Visual C++ 2005";  //^表示的就是一个句柄
   Console::WriteLine(str);
   }
```

“^”表示一个托管对象的句柄。根据 CLI 规范，句柄是对一个托管对象的应用，这样句柄和指针就不会混淆。句柄和指针的区别主要包括以下几项。

- 句柄用“^”表示，指针用“*”表示。
- 句柄可以被内存自动回收机制自动收回，指针不能被内存自动回收机制自动回收。
- 句柄，开发人员删除它是可选的；指针，开发人员必须将其删除。
- 句柄是对对象的引用，指针通常指向内存地址。
- 句柄类型是安全的，而指针却不是。
- 用“gcnew”关键字声明返回一个句柄，用“new”关键字声明返回一个指针。

2.1.2 类型的声明

CLR（Common Language Runtime）类型都是另加形容词前缀来进行描述的，比如原类型为 CLR 类型。

1）原类型

```
class man{…};
struct man{…};
```

2）CLR 类型

引用类型	ref class RefClass{…}；ref struct RefClass{…}；
值 类 型	value class Valclass{…}；value struct ValClass{…}；
接口类型	interface class IType{…}；interface struct IType{…}；

2.1.3 对代码编辑的改进

Visual Studio 2005 在代码编辑方面增加了新的特性，也有很多新的控制功能，使得开发更加简单。

- 行修正标记：通过此标记可以看到会话中正在编辑。已经编辑的行和临近的行都将被标记，如图 2-1 所示。
- 书签窗口：通过它用户可以管理书签。通过选择“编辑”→“书签”命令来对这一选项进行操作。
- 代码段：用户可以通过选择“工具”→“代码段管理器”命令来添加自定义的代码段。
- 自动恢复：这一特性可以恢复在过去 5 分钟内自动保存的修改。可以通过“工具”→“自动恢复”命令修改。

```
class CAboutDlg : public CDialog
{
public:
    CAboutDlg();

// 对话框数据
    enum { IDD = IDD_ABOUTBOX };

protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV 支持

// 实现
protected:
    DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
}
```

图 2-1　代码编辑器视图

Visual Studio 2005 新增的特性很多，这里就不一一说明了，在以后的使用中可逐渐发现。

2.2 VC能做的事情

Visual C++是一个功能强大的可视化软件开发工具。自 1993 年 Microsoft 公司推出以来，Visual C++已成为专业程序员进行软件开发的首选工具。

现在，Visual C++是广大工程技术人员最为熟悉的开发工具，同时，Visual C++又是进行网路和多媒体底层开发的主要工具，虽然其他开发语言，如 Java 等也可以进行网路多媒体开发，但是在实现速度上及灵活性方面，却远远不及 Visual C++。此外，更加重要的是，很多的标准库代码都是用 C++或者 C 语言来实现的。另外，数字编/解码技术对代码效率有很高的要求，对系统也有一定的依赖，所以本书选择了 Visual C++作为主要的开发工具。

2.2.1 生成传统的控制台应用程序

编写 Visual C++ 2005 的控制台应用程序有两种：分别为 Win32 控制台应用程序和 CLR 控制台应用程序，下面分别加以介绍。

1．生成传统的 Win32 控制台应用程序

用 Visual C++ 2005 编写程序的第一步是创建一个新项目。要生成传统的 Win32 控制台应用程序，步骤如下。

STEP 01 选择菜单栏上的“文件”→“新建”→“项目”命令（Ctrl+Shift+N），弹出“新建项目”对话框，如图 2-2 所示。

图 2-2 “新建项目”对话框

STEP 02 在弹出的对话框中选择“Visual C++”选项中的“Win32”子项，然后在右边的“模板”列表框中选择“Win32 控制台应用程序”选项，表示要创建一个 Windows 控制台应用程序。在该对话框下方的“名称”框内输入项目的名称，例如“MyProject”，在“位置”文本框内输入项目要保存的位置，也可以单击右边的“浏览”按钮修改项目的保存位置。“解决方案名称”在默认情况下与新建项目的名称相同，当然，也可以进行手工改动。这样，Visual C++ 2005 集成开发环境就建立了一个名为“MyProject”的目录，同时把所有的项目文件保存到该目录中。单击“确定”按钮进入下一步。

STEP 03 此时就会弹出一个“Win32 应用程序向导”对话框，第一个页面是“概述”页面，主要显示了当前的项目设置，如图 2-3 所示。单击“下一步”按钮，进入下一步。

STEP 04 此时会弹出“应用程序设置”界面，要求用户选择 Win32 应用程序的类型，如果要创建一个空项目，并且要手动输入源代码，则选中“附加选项”中的“空项目”复选框，否则选择“预编译头”复选框，然后再单击“完成”按钮，如图 2-4 所示。这样，Visual C++ 2005 就自动地创建了一个传统的 Windows 应用程序项目。

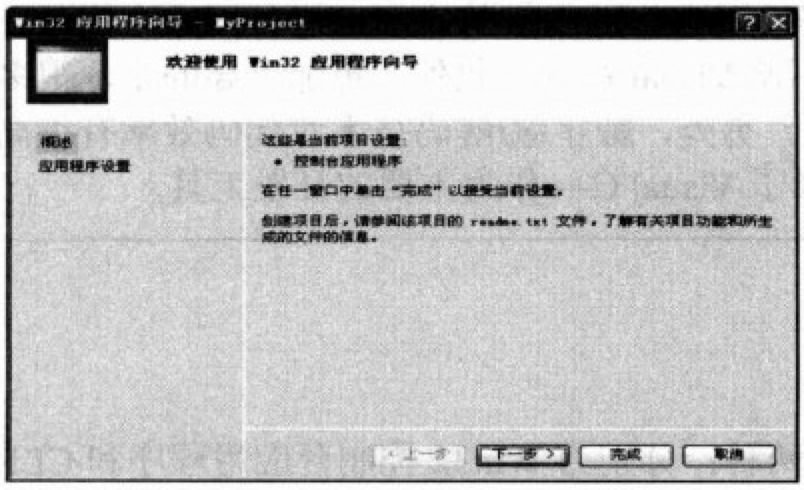


图 2-3 Win32 应用程序概述

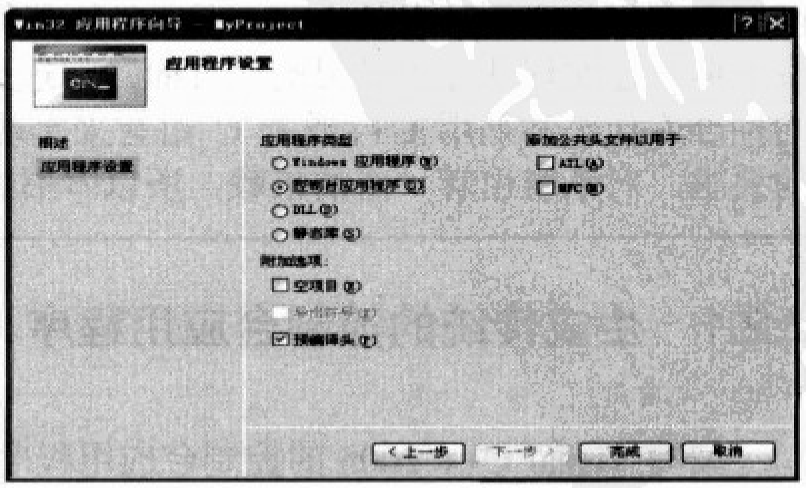


图 2-4 应用程序设置

STEP 05 至此就创建了一个 Win32 控制台应用程序项目，此时没有编辑任何代码，所以这一步要插入 C++源文件，如图 2-5 所示。

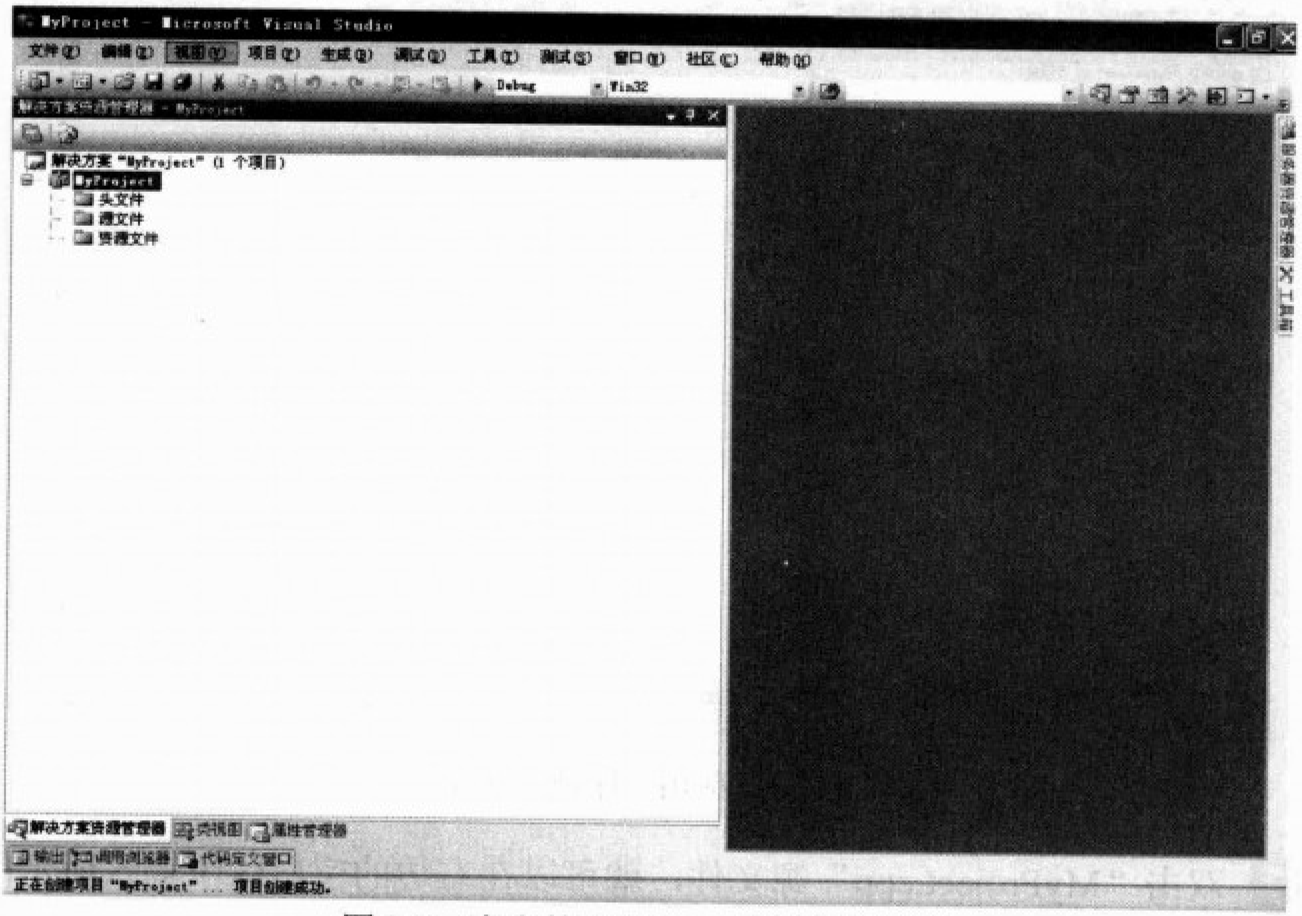


图 2-5 空白的 Windows 应用程序项目

用鼠标右键单击“解决方案资源管理器”面板中的“源文件”选项，如图 2-6 所示，选择“添加”→“新建项”命令，选择左侧列表框中的“代码”选项，在右侧的“模板”中选中“C++文件（.cpp）”选项，在“名称”文本框中输入文件名。如果不输入 C++类型文件的后缀名.cpp，生成的应用程序也会自动将文件后缀名添上，生成的文件“MyProject.cpp”将保存到当前项目的目录下，然后单击“添加”按钮，如图 2-7 所示。

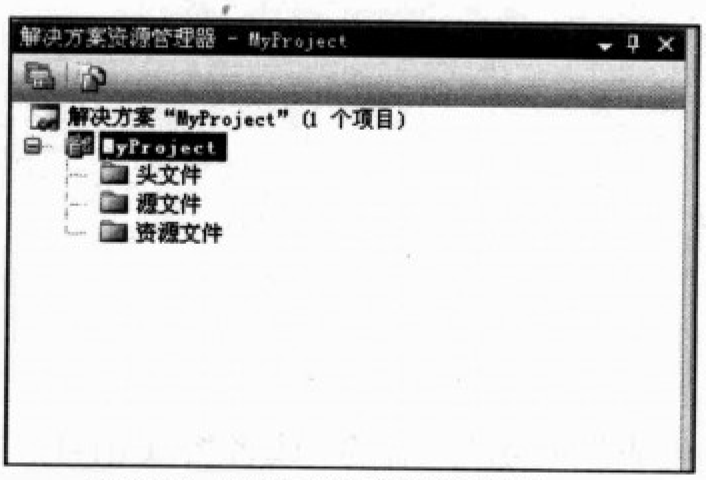


图 2-6 解决方案资源管理器

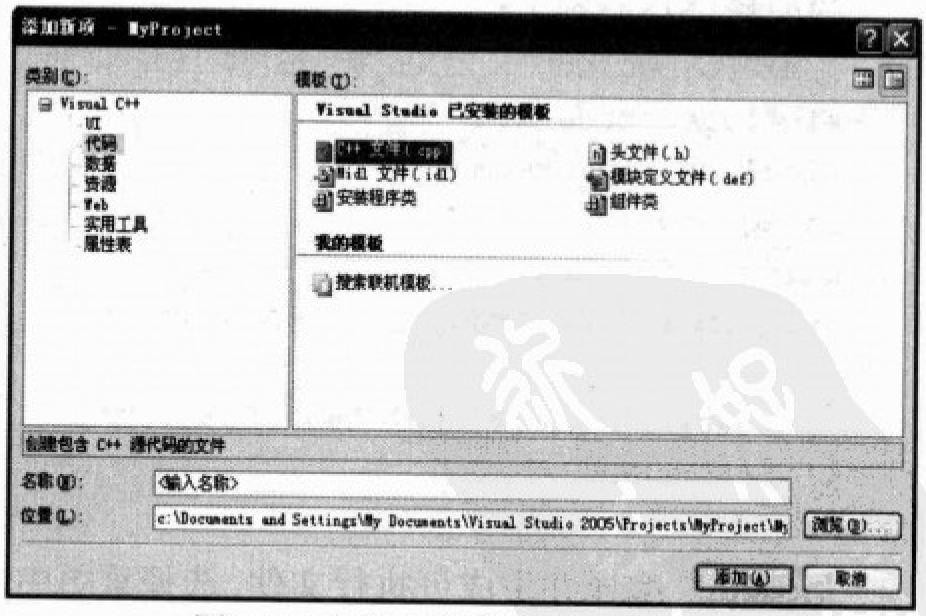


图 2-7 在源文件中新建一个 C++源文件

STEP 06 然后就开始编写源文件了，这时 Visual C++2005 的“解决方案资源管理器”面板中列出了所有的文件，而“类视图”则列出了程序的所有类，“属性管理器”则列出了程序的项目的所有属性。此时选中“解决方案资源管理器”面板中的“源文件”选项就能看到刚刚新加入到项目中的文件“MyProject.cpp”，如图 2-8 所示。

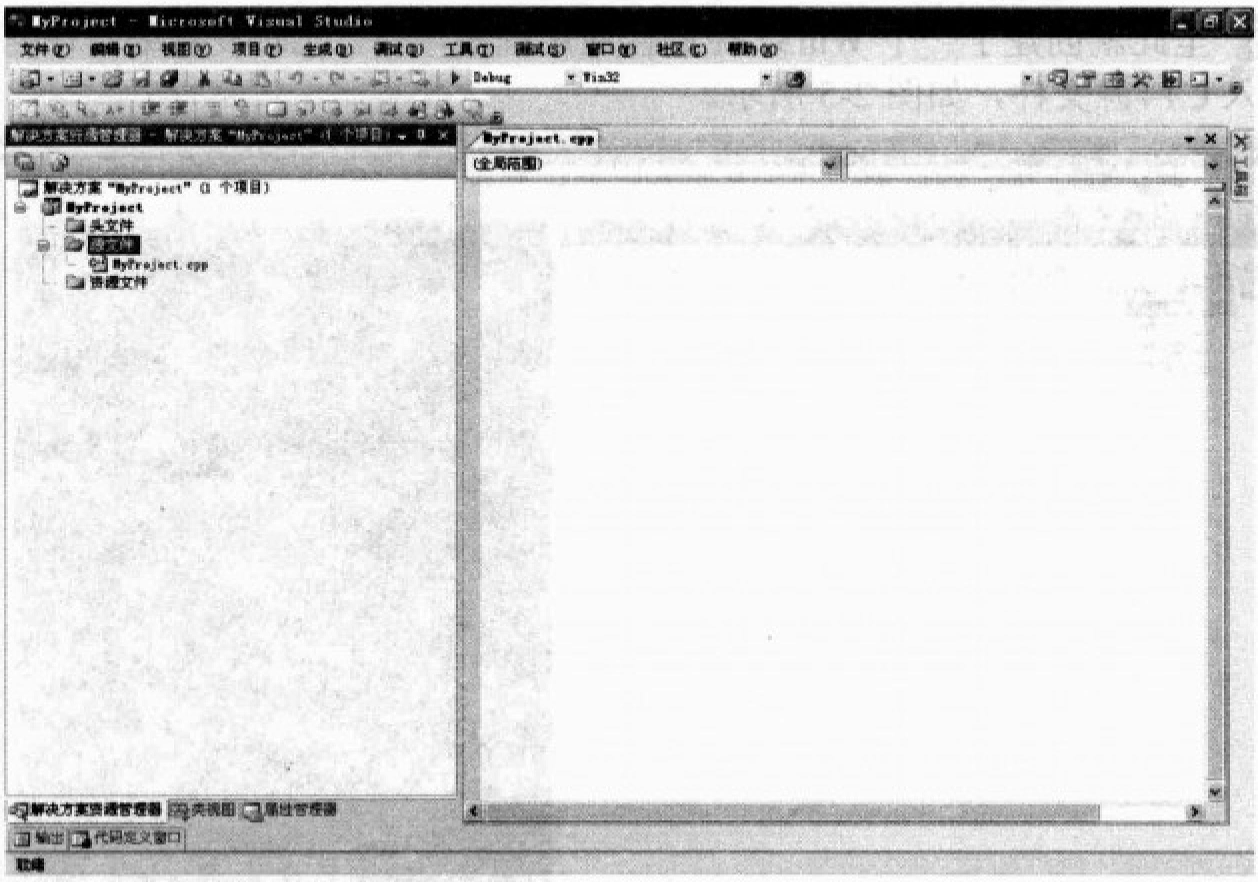


图 2-8　应用程序项目界面

STEP 07 双击“MyProject.cpp”源文件，就可以在右边的空白区中输入所示的源代码：

```
//MyProject.cpp : 定义控制台应用程序的入口点
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
    return 0;
}
```

然后修改代码如下：

```
//MyProject.cpp : 定义控制台应用程序的入口点。
#include "stdafx.h"
#include <iostream>
using std::cin;
using std::cout;
int _tmain(int argc, _TCHAR* argv[])
{
   cout<<"Hello,Visual C++ 2005!.\n";
   return 0;
}
```

STEP 08 编译并生成可执行文件。选择菜单中的“生成”→“编译”命令，或者按“Ctrl+F7”组合键，编译该解决方案并生成可执行文件。

STEP 09 执行可执行文件。选择“调试”→“开始执行（不调试）”命令，或者按“Ctrl+F5”组合键即可执行程序，运行程序的结果如图 2-9 所示。

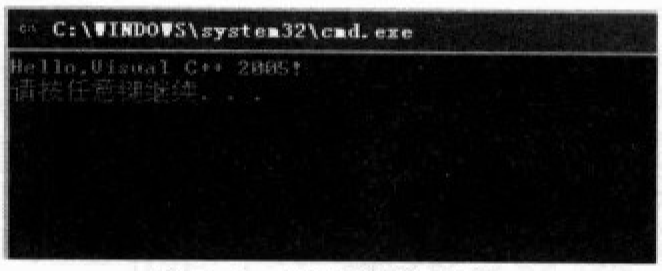


图 2-9 程序运行结果

2. 生成 CLR 控制台应用程序

生成 CLR 控制台应用程序的步骤如下。

STEP 01 选择菜单中的“工具”→“新建”→“项目”命令，打开“新建项目”对话框，如图 2-10 所示。

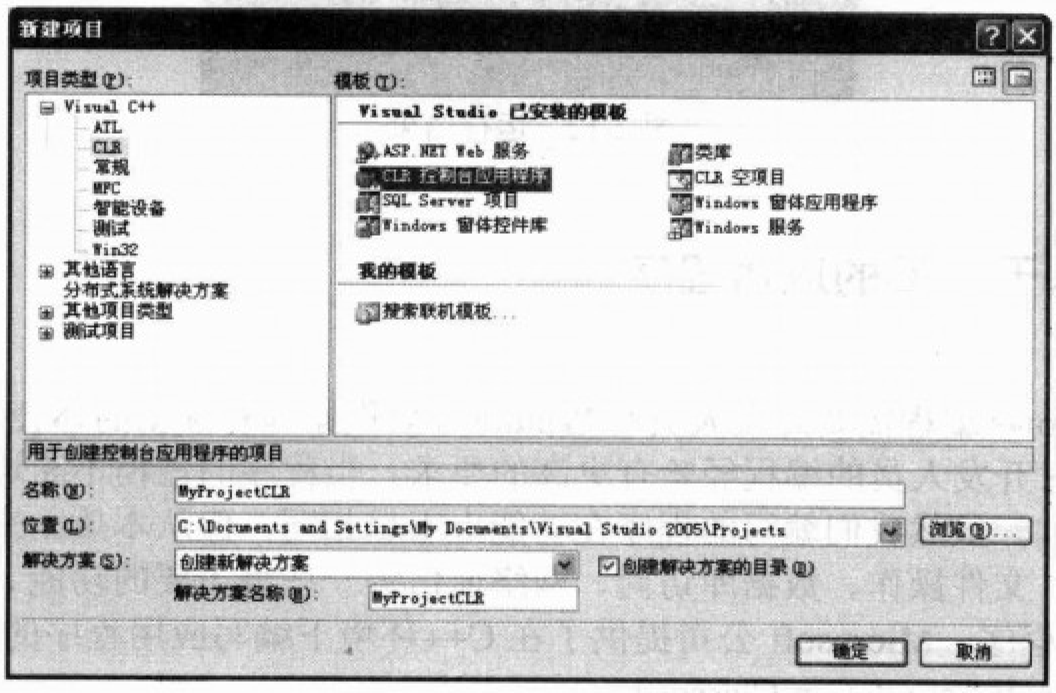


图 2-10 “新建项目”对话框

STEP 02 在左侧的面板列表中选择“Visual C++”中的“CLR”选项，在右侧的“模板”面板中选择“CLR 控制台应用程序”选项。输入项目名“MyProjectCLR”，在“位置”文本框中输入项目要保存的位置，其他选择默认状态。单击“确定”按钮，就完成了项目的创建。

STEP 03 在弹出的界面中，左侧是“解决方案资源管理器”面板，如图 2-11 所示。

STEP 04 在左侧“解决方案资源管理器”面板中选择“MyProjectCLR.cpp”文件，双击并打开，默认的代码如下所示。

图 2-11 解决方案资源管理器

```
// MyProjectCLR.cpp: 主项目文件
#include "stdafx.h"
using namespace System;
int main(array<System::String ^> ^args)
{
   Console::WriteLine(L"Hello,Visual C++ 2005!");
   return 0;
}
```

STEP 05 运行程序结果如图 2-12 所示。

图 2-12 运行结果

2.2.2 生成基于MFC的应用程序

传统的 Windows 编程需要开发人员对 Windows 编程原理有深刻的体会，传统的 Windows API 编程模式也对开发人员的编程经验有更高的要求。当程序的规模不断增大时，出错的几率也会大幅度增加。虽然我们编写的程序在功能上不尽相同，但从本质上来说，都可以归结为用户界面设计、文件操作、数据库访问、网络通信等一些最主要的功能。为了方便开发人员编写 Windows 程序，Microsoft 公司提供了在 C++环境下编写应用程序的框架 MFC，即微软基础类（Microsoft Foundation Classes）。

MFC 是用 C++编写的一套基础类。MFC 用面向对象的技术对 95%以上的 API 进行了包装，因此功能非常强大。利用 MFC 提供的类，我们可以很方便地管理窗口、菜单、对话框、文档和视图，执行基本的输入/输出操作等。此外 MFC 还提供了应用程序框架，利用面向对象的技术，如重载、派生等，可以扩展应用的功能。

Visual C++ 2005 的 MFC 中包括的类，涉及用户界面的设计、文件的操作、数据库的访问、COM、ActiveX 等众多方面。在设计程序时，可以通过继承类库中的已有类产生更专业的、更强大的类库。正是由于 MFC 编程方法充分利用了面向对象技术的优点，使得编程时极少需要关心对象方法的实现细节，同时类库中的各种类的强大功能足以完成程序的绝大部分所需功能，因此应用程序的开发人员所需要编写的代码大大减少，这就增加了程序的可调试性。

1. MFC 中的类

MFC 中提供了很多的类，这些类被分组，每组类提供一种功能。有些类封装 Win32 应用程序编程接口（API）的大部分内容。有些类则封装文档/视图等应用程序概念和应用程序本身。还有一些类封装 OLE（Object Liinking and Embedding，对象链接和嵌入）功能及 ODBC（开放式数据库连接）和 DAO 数据访问功能，几乎覆盖了编程的各个方面。下面列出了 MFC 中的所有类（按派生关系），如图 2-13 所示。

图 2-13　MFC 中的所有类

从图 2-13 可以看出，MFC 中的类可以分为两大部分：一部分类是从 CObject 类派生而来的，另一部分不是从 CObject 类派生而来的。

下面，就简单了解一下 MFC 的类库。在以后的音频、视频开发中会用到其中的部分类。MFC 类库中的大部分类都派生自 CObject 类，如图 2-14 所示。

但是，在 MFC 中也有一些类并不是从 CObject 类派生而来的，如图 2-15 所示。

MFC 类库中的绝大部分类都是从 CObject 类派生而来的。这意味着相当多的 MFC 类都继承了 CObject 类的特性。所以，理解 CObject 类就成为了理解 MFC 类库的关键。

2．CObject 类

CObject 类是 Microsoft 基础类库中主要的基类。它不仅用做库类，例如 CFile 和 CObList 的根，而且也是自己编写的类的根。CObject 类提供了非常重要的基本功能，其中包括以下几项。

- 基本功能：不支持运行时类信息或序列化，但是包含诊断内存管理。
- 运行时类信息和动态创建支持。
- 对象诊断输出支持。
- 与集合类兼容。

图 2-14　从 CObject 类派生的 MFC 类

图 2-15　所有不从 CObject 类派生的 MFC 类

先看一下 CObject 类的声明，对于 MFC 内部的类来说，清楚地知道 CObject 类都提供了哪些函数及这些函数如何工作是很重要的。下面程序列出了 CObject 类的声明。

```
//CObject类声明
Class CObject
{
public:
    virtual CRuntimeClass* GetRuntimeClass() const;
virtual ~CObject()=0;  // 虚构造函数是必需的
//诊断分配
    void* operator new(size_t,void* p);
    void* operator new(size_t,nSize);
    void operator delete(void* p);
#ifdef _Debug
    void* operator new(size_t nSize,LPCSTR lpszFileName, int nLine);
#endif
```

```
protected:
   CObject();
private:
   CObject(const CObject& objectSrc);                //不实现
   void operator=(const CObject& objectSrc);        //不实现
//属性
public:
   BOOL IsSerializable() const;                     //支持持久性
   BOOL IsKindOf(const CRunTimeClass* pClass) const;
   virtual void Serialize(CArchive& ar);            //支持持久性
//诊断支持
   virtual void AssertValid() const;
   virtual void Dump(CDumpContext& dc) const;
//实现
   public:
      static const CRunTimeClass classCObject;      //支持运行时类型检查
};
```

CObject 的声明中运用了许多 C++的小技巧。首先，析构函数是虚函数，这样就可以通过多态机制调用析构函数，也就是 MFC 打算自己创建和删除很多从 CObject 类派生的类，这样在不需要调用正确的删除操作符和析构函数的情况下，它就能删除一个对象。

Object 类中的复制构造函数 Cobject（const CObject& objectSrc）和赋值操作符“=”都被声明为私有函数，目的就是便 Object 类的派生类对象不能通过复制构造函数的方式产生新对象，这是因为复制构造函数和赋值都要调用基类的复制构造函数和赋值操作符。

3. 常见的 MFC 类

通常类的定义和实现是 MFC 的核心，MFC 中有 200 多个类，而 Winows 程序中的每一类功能，都有一个具体的 MFC 类与它对应。下面只对一些常用的类进行简要的介绍。

1）CCmdTarget 类

CCmdTarget 类是由 CObject 类直接派生而来的，如图 2-16 所示，CCmdTarget 类又称为消息映射类，它负责将消息发送到能够响应这些消息的对象。CCmdTarget 类是所有能进行消息映射的 MFC 类的基类，从 MFC 的层次图可以看出，CWnd、CView、CFrameWnd、CWinApp、CwinThread、CDocument 和 CdocTemplate 等类都是从 CCmdTarget 类派生而来的。CCmdTarget 类的功能主要包括消息发送、设置光标和支持自动化等。

- 消息发送：CCmdTarget 是所有接收窗口消息和命令的对象的基类。MFC 为每个 CCmd Target 类的派生来创建一个称为消息映射表的静态数据结构，消息映射表就是将窗口与处理消息的类成员函数关联起来。消息映射表数据结构和消息映射宏是消息映射系统的另外两个重要的方面。
- 设置光标：在 CCmdTarget 类中有 3 个成员函数用于设置光标。成员函数 BeginWaitCursor()用于将光标改为沙漏形状，提示程序正在进行某种操作。当操作完成时，函数 EndWaitCursor()将光标改回调用函数 BeginWaitCursor()之间的形状。当处于等待状态时由于某些外部操作（如显示一个信息对话框后）改变了光标的形状，函数 RestoreWaitCursor()将光标还原为等待状态。

- 支持自动化：自动化是一种 OLE 特性。支持程序通过 COM 接口进行交互操作，自动翻译 COM 接口的成员函数。CCmdTarget 类支持自动化的方法是调用函数 EnableAutomation()、FromIDispatch()、GetDispatch()、IsResultExpected() 和 OnFinalRelease()等。

```
CObject
   └CCmdTarget
```

图 2-16　CCmdTarget 类的派生关系

2）CWnd 类

CWnd 类是由 CCmdTarget 类直接派生而来的，它是 MFC 中所有窗口类（如 CFrameWnd 类、CView 类）的基类，该类及其派生类的实例是一个窗口。CWnd 类代表了 MFC 中最基本的图形用户界面对象，它是功能最丰富、成员函数最多的 MFC 类。CWnd 类的派生关系如图 2-17 所示。

按照 Windows 的机制，窗口是一个已创建属性（如标题、位置和大小等）和行为（如窗口函数 WndProc）的窗口类的对象。多数窗口都有一个代表它的图形界面，能接收、处理系统消息和窗口消息，响应用户输入操作。窗口的实例包括应用程序主窗口、对话框和各种控件等。CWnd 类提供的功能包括创建新窗口类、创建窗口及子窗口、获取窗口、管理窗口、访问窗口及控件、控制窗口光标、创建和使用句柄和支持工具提示等，具体成员函数的功能参考 MSDN 帮助。

```
CObject
   └CCmdTarget
        └CWnd
```

图 2-17　CWnd 类的派生关系

3）CrFameWnd 类

CFrameWnd 类是由 CWnd 类派生而来的，它是单文档应用程序主框架窗口的基类，也是其他框架窗口类的基类，其派生关系如图 2-18 所示。CFrameWnd 类主要用来管理一个框架窗口，CFrameWnd 类的一个对象是一个框架窗口，包括边框、标题栏、菜单、最大化按钮、最小化按钮和一个激活的视图。CFrameWnd 类用于单文档框架窗口，对于多文档框架窗口，使用它的两个派生类为 CMDIFrameWnd 和 CMDIChildWnd。CMDIFrameWnd 类用于 MDI（多文档界面）主窗口，CMDIChildWnd 类用于 MDI 的子窗口。

CFrameWnd 类提供了一些成员函数用于获得和设置文档、视图、图文框、标题栏和状态栏等。常用的函数有 GetActiveDocument()得到当前文档的指针、GetActiveView()得到当前视图的指针、GetTitle()得到框架窗口的标题、SetTitle()设置框架窗口的标题、SetActiveView()激活一个视图、GetActiveFrame()得到当前框架窗口的指针，以及 SetMessageText()设置状态栏文本。

```
CObject
   └CCmdTarget
        └CWnd
             └CFrameWnd
```

图 2-18　CFameWnd 类的派生关系

4）CView 类

CView 类也是从 CWnd 类派生而来的，其派生关系如图 2-19 所示。CView 类是 MFC 中的一个基本类，也是其他 MFC 视图类和用户视图派生类的基类。视图用来显式文档，文档为视图提供数据，当一个文档中的数据改变时，可以通过调用视图的成员函数 OnUpdate()来刷新视图，也可以通过调用其基类 CWnd 的成员函数 Invalidate()或 InvalidateRect()刷新视图。

CObject
　└CCmdTarget
　　　└CWnd
　　　　　└CView

图 2-19　CView 类的派生关系

5）CDocument 类

CDocument 类是 CCmdTarget 类的派生类，其派生关系如图 2-20 所示。CDocument 类在应用程序中作为文档类的基类，它代表用户当前所操作的文档。CDocument 类的主要功能是把数据处理的任务从界面显示中分离出来，同时提供一个与视图类交互的接口。

CDocument 类支持标准的文件操作，如创建、打开和存储一个文档等。一个应用程序可以支持多种类型的文档，每种类型的文档都通过一个文档模板对象建立与各自视图的联系。该类的常用成员函数有 OnNewDocument()建立新的文档、OnOpenDocument()打开文档、OnCloseDocment()关闭文档、OnSaveDocument()保存文档和 SaveModified()设置文档修改标志，以及 UpdateAllView()通知所有视图问题已被修改。

CObject
　└CCmdTarget
　　　└CDocument

图 2-20　CDocument 类的派生关系

我们用 Visual C++ 2005 创建 Windows 应用程序通常有以下几种结构。

- 传统的控制台应用程序。
- 基于框架窗口的应用程序。若应用程序仅仅需要最小的用户界面和简单的窗口结构，则可以采用基于框架窗口的应用程序。
- 基于对话框的应用程序。
- 基于文档/视图的应用程序。文档/视图应用程序结构比较复杂，功能也比较强。

4．用 MFC 来生成 Windows 应用程序

下面用 MFC 来生成 Windows 应用程序。

STEP 01 打开 Visual C++ 2005，选择“文件”→“新建”→“项目”命令，在弹出的对话框中选择“Visual C++”列表中的“MFC”选项，然后在“模板”选项列表中选择“MFC 应用程序”选项，输入项目名称“MyProjectMFC”，设置文件的保存路径，单击“确定”按钮。

STEP 02 弹出“MFC 应用程序向导”对话框，如图 2-21 所示。此对话框显示的是对当前项目的设置。单击“下一步”按钮，会弹出“应用程序类型”设置界面，如图 2-22 所示。

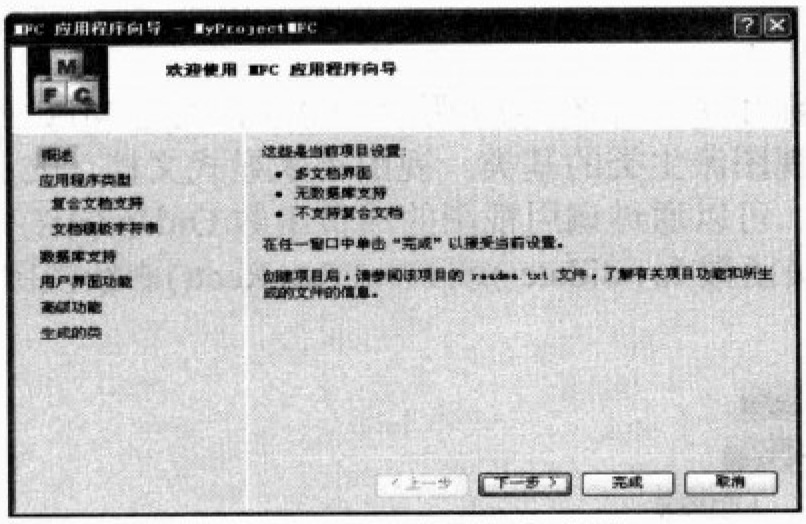


图 2-21　MFC 应用程序向导

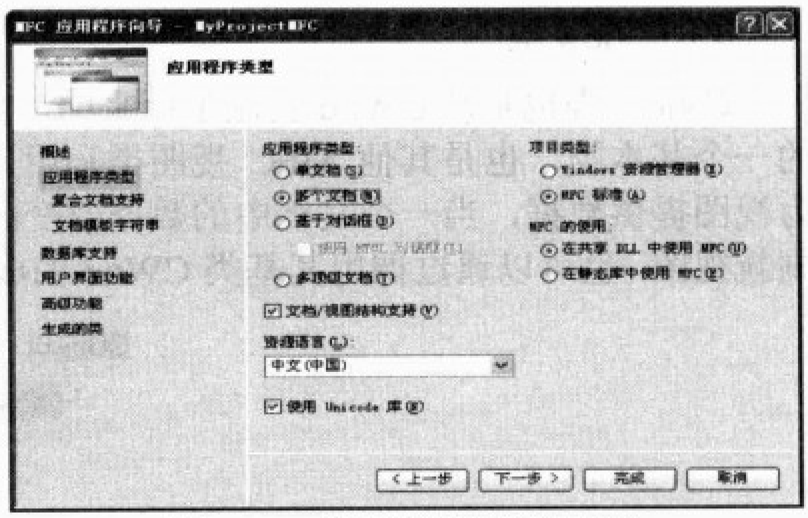


图 2-22　应用程序类型的选择

STEP 03 在对话框中，根据需要选择“应用程序类型”、“项目类型”和“资源语言”中的选项，此时使用默认值。然后单击“下一步”按钮，弹出“复合文档支持”界面，如图 2-23 所示。在“复合文档支持”界面中，选择默认状态，单击“下一步”按钮，出现“文档模板字符串”界面，如图 2-24 所示。

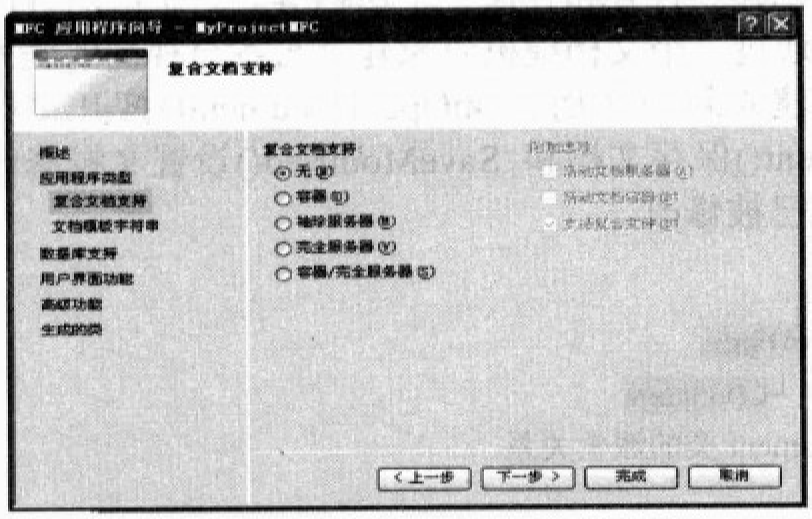


图 2-23　复合文档支持

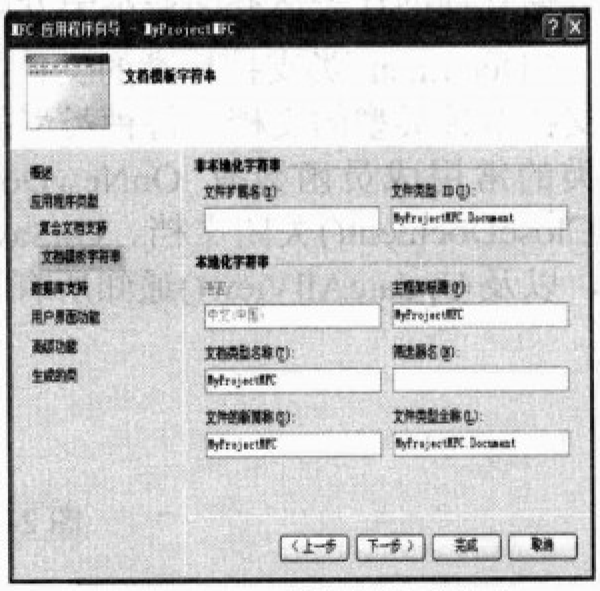


图 2-24　文档模板字符串

STEP 04 在“文档模板字符串”界面中，选择默认状态，单击“下一步”按钮，弹出“数据库支持”界面，如图 2-25 所示。

STEP 05 在“数据库支持”界面中，选择默认状态，单击“下一步”按钮，弹出“用户界面功能”界面，如图 2-26 所示。

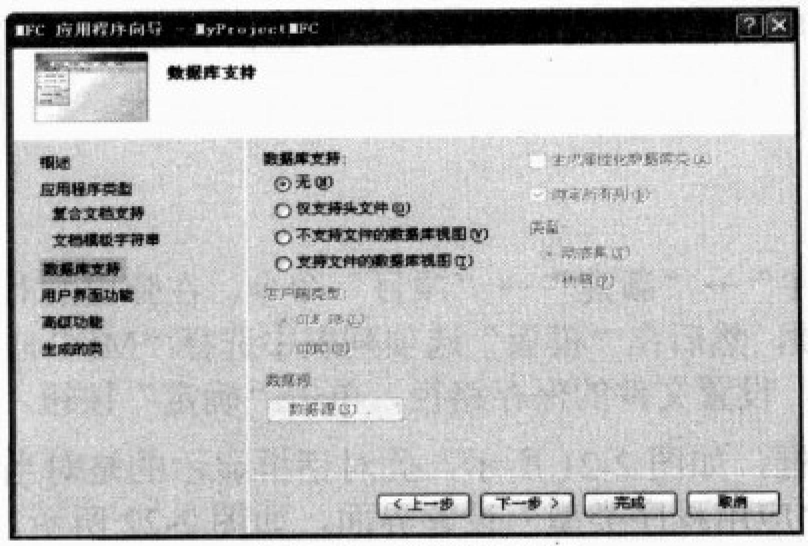


图 2-25　数据库支持

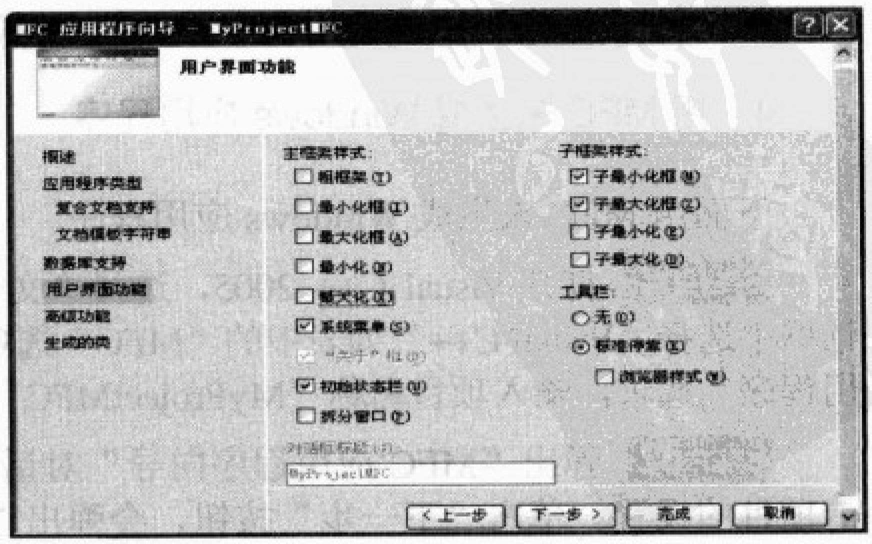


图 2-26　用户界面功能

STEP 06 根据需要选择“主框架样式”、“对话框标题”、“子框架样式”和“工具栏”中的选项，单击“下一步”按钮，出现“高级功能”界面，如图 2-27 所示。在“高级功能”界面中选择默认状态，单击“下一步”按钮，弹出如图 2-28 所示的“生成的类”界面。

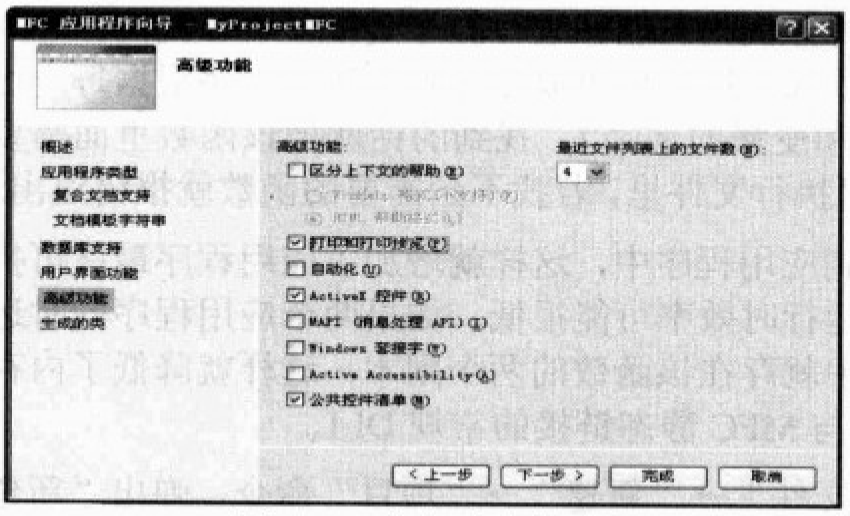


图 2-27　高级功能的选择

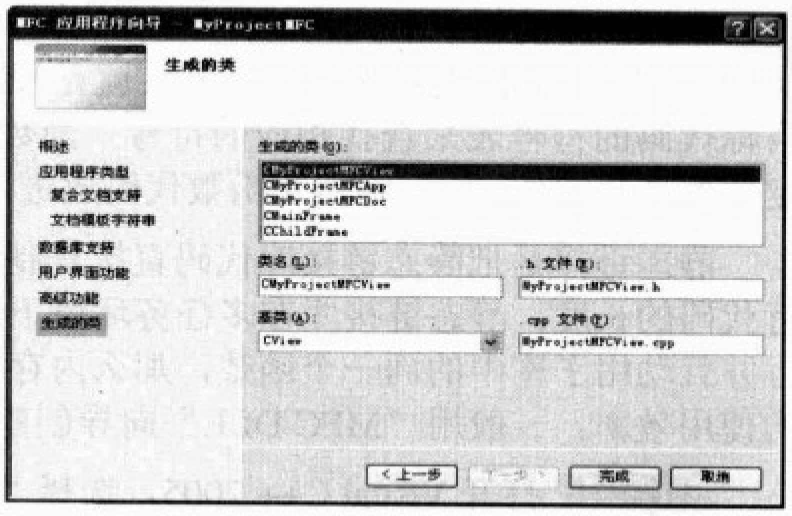


图 2-28　生成的类的设置

STEP 07 单击“完成”按钮，完成 MFC 应用程序项目的创建，会出现一个空白的 MFC 应用程序项目界面，如图 2-29 所示。弹出“解决方案资源管理器”对话框，如图 2-30 所示。

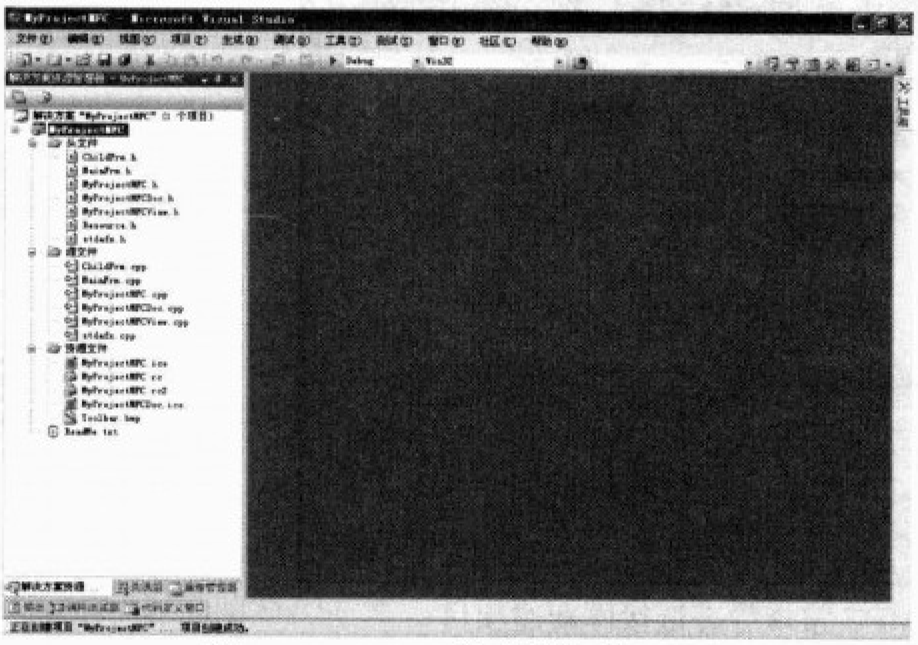


图 2-29　MFC 应用程序项目界面

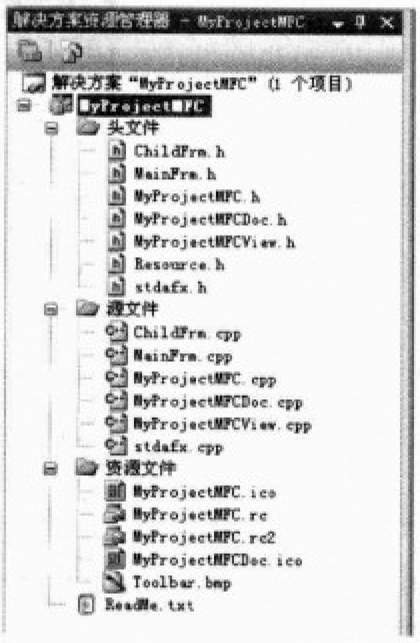


图 2-30　解决方案资源管理器

STEP 08 最后运行该程序，结果如图 2-31 所示。至此，利用 MFC 向导创建了一个简单多文档的 MFC 应用程序。

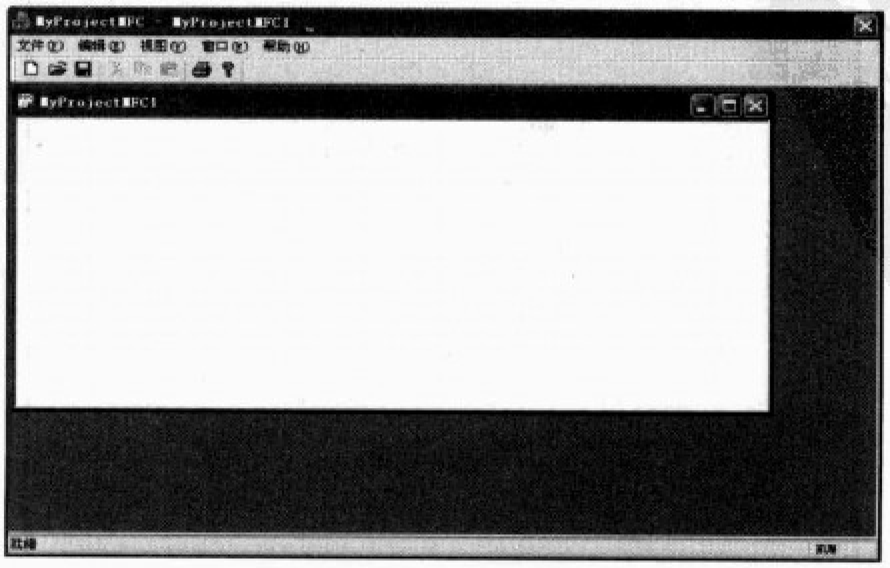


图 2-31　运行结果

2.2.3 生成静态链接库

早期，库的组织形式相对简单，里面的目标代码只能够进行静态链接，所以我们称之为“静态库”。静态库的结构简单，其实就是把原来的目标代码放在一起，链接程序根据每一份目标代码的符号表来查找相应的符号（函数和变量的名字），找到的话就把该函数里面需要定位的进行定位，然后将整块函数代码放进可执行文件里，若找不到需要的函数就报错退出。

静态链接是把静态链接库代码直接复制到应用程序中，这样就增加了应用程序最终可执行代码的长度。静态链接库在多任务环境下运行时效率可能很低，如果两个应用程序同时运行并且动用了库中的同一个函数，那么内存中就存在该函数的两个副本，这样就降低了内存的使用效率。一般用“MFC DLL”向导创建与 MFC 静态链接的常规 DLL。

STEP 01 打开 Visual C++ 2005，选择“文件”→“新建”→“项目”命令，弹出“新建项目”对话框，在左侧面板“项目类型”中选择“MFC”选项，在“模板”列表中选择“MFC DLL”选项，并在“名称”文本框中输入要生成的 DLL 的文件名“MyProjectLIB”。在“位置”文本框中输入或选择要保存的路径，如图 2-32 所示。

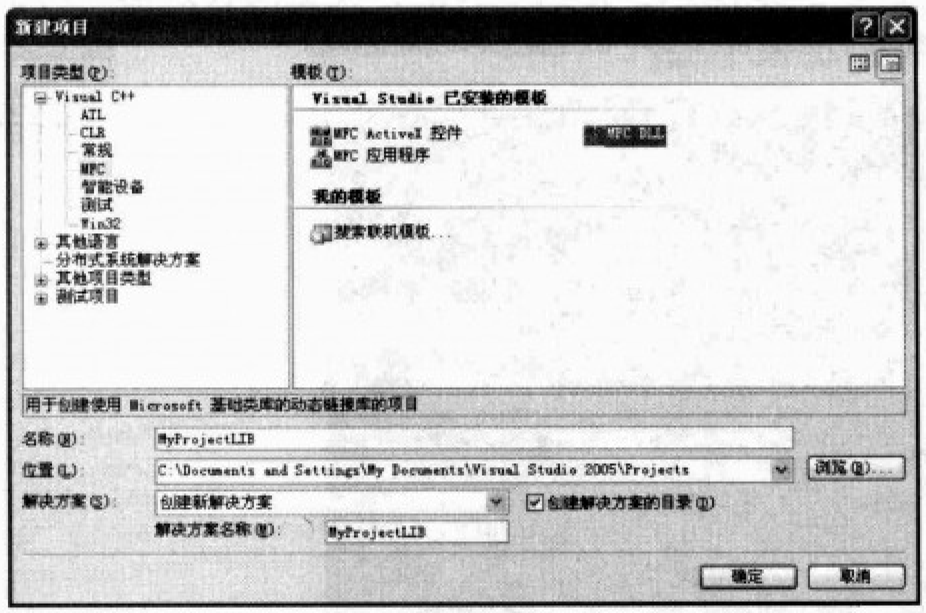


图 2-32 MFC 常规静态链接库新建项目

STEP 02 单击“确定”按钮，进入“MFC DLL 向导”对话框，如图 2-33 所示。在“DLL 类型”中，选择“带静态链接 MFC 的规则 DLL”单选按钮，其他可采用默认值。

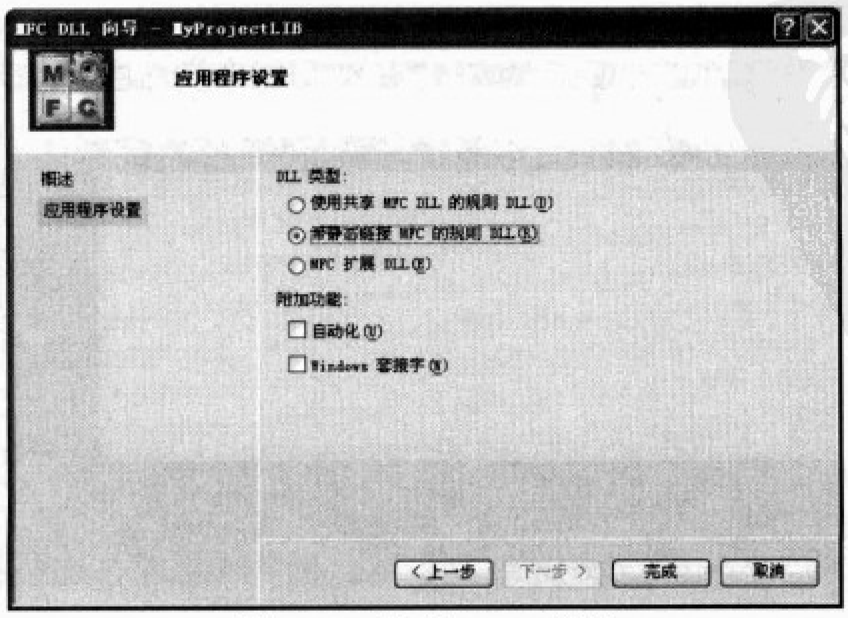


图 2-33 选择 DLL 类型

STEP 03 单击“完成”按钮，此时 MFC DLL 向导可自动完成静态链接库的创建，而在指定的目录下生成 DLL 项目所必需的全部文件。生成的静态链接库界面如图 2-34 所示。

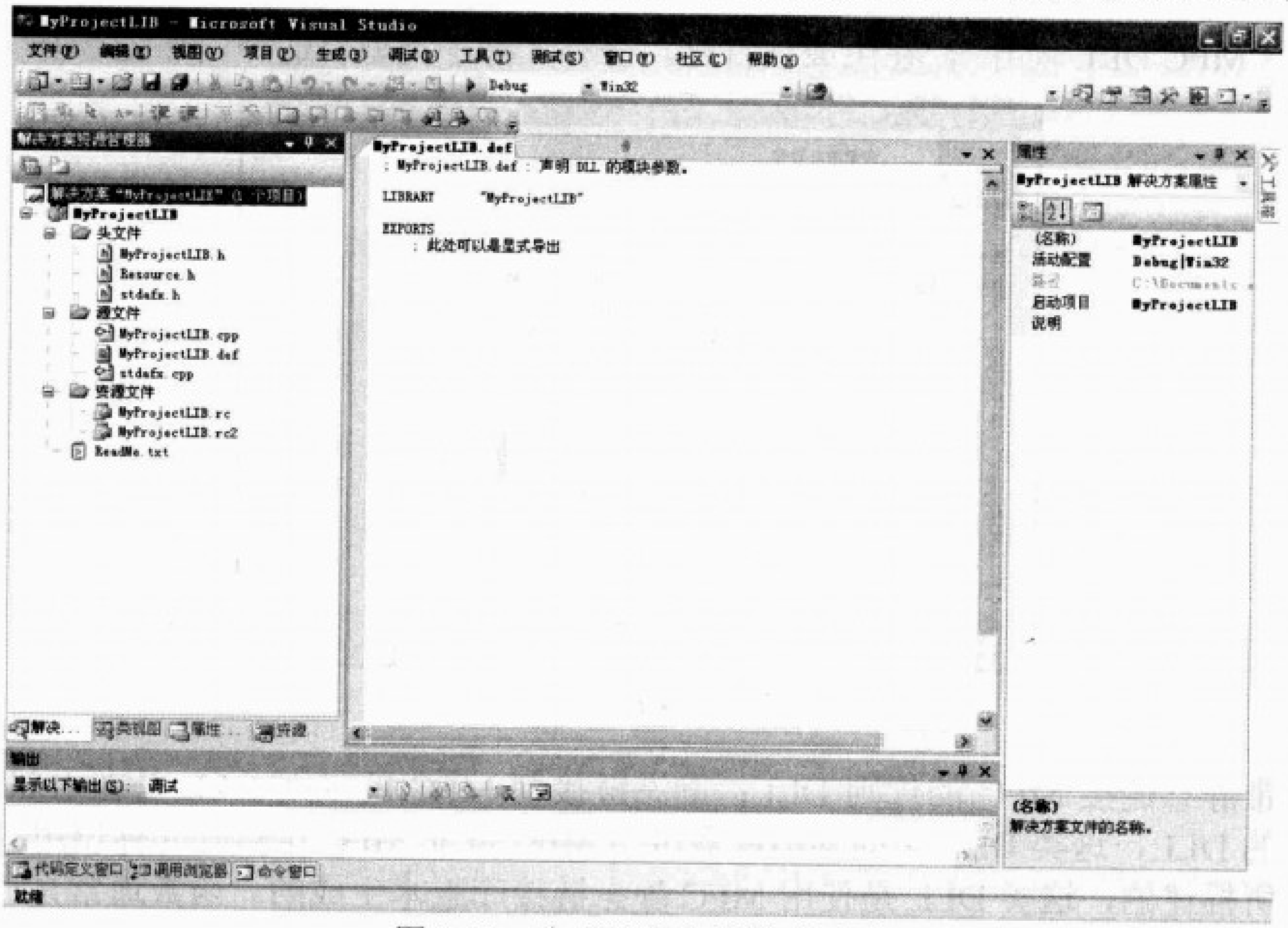


图 2-34　生成的静态链接库界面

然后我们就可以在代码编辑区编写代码，最后编译生成静态链接库。

2.2.4　生成动态链接库

为了提高系统内存资源的使用效率，每一个应用程序应该尽量少占用内存空间。在多任务的环境下，同时运行的多个应用程序有时要调用相同的函数，如标准的数学函数和 Windows 环境下的窗口函数。当有多个应用程序同时运行并且都调用同一个函数时，这些应用程序应该可以共享这个函数在内存中的执行代码，这样就减少了内存的占用。动态链接库便是这一设想的具体实现。

动态链接库（Dynamic Link Library，DLL）是 Microsoft Windows 最重要的组成要素之一。它是由函数对象及资源等组成的二进制库文件，此库文件被多个进程或者应用程序使用。也就是说，这是其他应用程序在运行时可以调用的一组服务和函数的模块。

从结构上看，一个 DLL 就是一个 C++源文件，它包含互相之间可以或者不可以进行相互作用的几个函数。此外，DLL 没有对 API 函数进行限制，如字符、对话框等。它可以登记一个新的窗口类，生成窗口或者只是提供一组函数，以支持和完成应用程序。

我们知道，所有的 Windows 应用程序都离不开动态链接库，因为 Windows API 函数就是以动态链接库的形式提供的。DLL 是动态链接库的常用文件扩展名，另外 DRW 及部分 EXE 后缀的文件也是动态链接库。

1．动态链接库的类型

使用“MFC DLL 向导”，在生成项目时，会有 3 种子类型供选择，如图 2-35 所示。

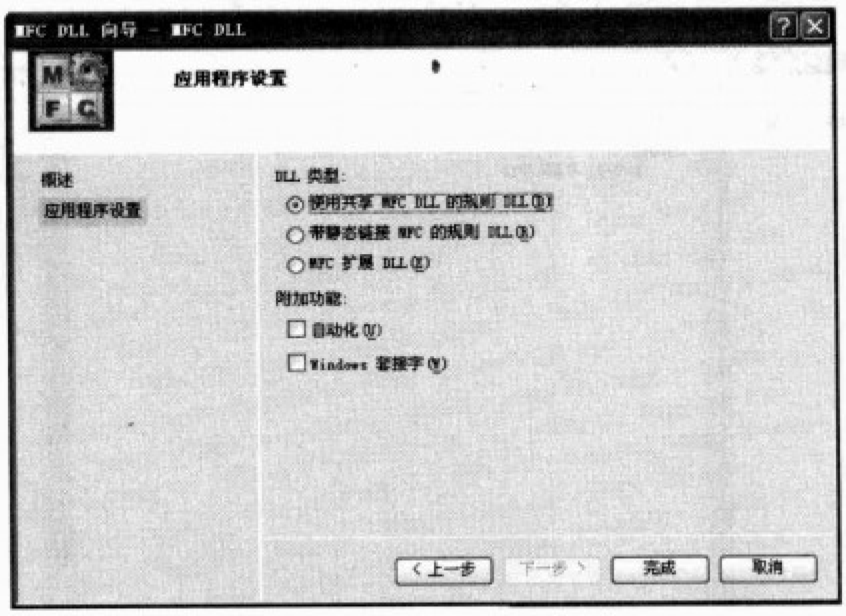


图 2-35　MFC DLL 向导

- 带静态链接 MFC 的规则 DLL：静态链接到 MFC 的规则 DLL 是在内部使用 MFC 的 DLL，这类 DLL 中的导出函数可由 MFC 或非 MFC 可执行文件调用。正如名称所描述的，这类 DLL 是使用 MFC 静态链接库版本生成的。函数通常是通过标准 C 接口从规则 DLL 导出的。
- 使用共享 MFC DLL 的规则 DLL：动态链接到 MFC 的规则 DLL 是在内部使用 MFC 的 DLL，这类 DLL 中的导出函数可由 MFC 或非 MFC 可执行文件调用。正如名称所体现的，这类 DLL 是使用 MFC 动态链接库版本（也称做 MFC 共享版本）生成的。函数通常是通过标准 C 接口从规则 DLL 导出的。
- MFC 扩展 DLL：MFC 扩展 DLL 是通常实现从现有 Microsoft 基础类库类派生的可重用类的 DLL。扩展 DLL 是使用 MFC 动态链接库版本（也称做共享 MFC 版本）生成的。只有用共享 MFC 版本生成的 MFC 可执行文件（应用程序或规则 DLL）才能使用扩展 DLL。使用扩展 DLL，可以从 MFC 派生新的自定义类，然后将此“扩展”版本的 MFC 提供给调用 DLL 的应用程序。

扩展 DLL 也可用于在应用程序和 DLL 之间传递 MFC 派生的对象。与已传递的对象关联的成员函数存在于创建对象所在的模块中。由于在使用 MFC 的共享 DLL 版本时正确导出了这些函数，因此可以在应用程序和它加载的扩展 DLL 之间随意传递 MFC 或 MFC 派生的对象指针。

规则 DLL 可以被 MFC 应用和非 MFC 应用使用。如果选择静态链接到 MFC 库，那么规则 DLL 中将包含 MFC 库代码副本。

扩展 DLL 智能动态链接到 MFC 库，但仅限于在 MFC 应用中使用。与规则 DLL 相比，扩展 DLL 的一个好处是，它可以定义从现有 MFC 类派生的子类并输出这些类。其他 MFC 应用（包括扩展 DLL、规则 DLL）可以使用这些类，就像是 MFC 库实现了这些类一样。

2．动态链接库的加载

将动态链接库链接到应用程序分为隐式链接（Implicit Linking）和显式链接（Explicit

Llinking）两种方式。

1）隐式链接

隐式链接就是指在动态链接库的应用程序运行时，由 Windows 操作系统将要使用的 DLL 自动加载到应用程序。当链接静态时，而在编写代码过程中使用动态链接库中的函数时，只要调用本地函数即可，也就是说，在已经包含了函数的.h 文件，直接引用函数，在链接的时候将.lib 文件链接进去。当隐式链接后，.exe 程序一加载，Windows 自动先查找.exe 文件的当前目录有没有.dll，如果没有，则按搜索路径查找，如果搜索路径也没有，则会弹出对话框报错，提示找不到.dll 文件，若显示加载在 LoadLibrary 函数返回的句柄为 NULL，则说明找不到.dll，否则就是.dll 文件加载进内存的句柄。

如果要对一个动态链接库隐式链接，首先必须在每个使用导出函数的源文件中将相应的头文件包含进来，然后必须将动态链接库的库文件（.lib）导入到项目中。

在最后，还要将 DLL 存放在相应的路径下，因为 DLL 的加载是由操作系统完成的，所以一般可将 DLL 放在应用程序所在的路径下或者系统目录下，这样就可以将一个动态链接库文件隐式地加载到应用程序中。应用程序在使用动态链接库的导出函数时，可以像使用自己定义的函数那样直接调用，不需要另外做任何说明。

采用隐式链接方式，具体方式就是：程序员在建立一个 DLL 文件时，链接程序会自动生成一个与之对应的 LIB 导入文件。该文件包含了每一个 DLL 导出函数的符号名和可选的标识符，但是并不含有实际的代码。LIB 文件作为 DLL 的替代文件被编译到应用程序项目中。当程序员通过静态链接方式编译生成应用程序时，应用程序中的调用函数与 LIB 文件中导出符号相匹配，这些符号或标识符进入到生成的 EXE 文件中。LIB 文件中也包含了对应的 DLL 文件名（但不是完全的路径名），链接程序将其存储在 EXE 文件内部。当应用程序运行过程中需要加载 DLL 文件时，Windows 根据这些信息发现并加载 DLL，然后通过符号名或标识符实现对 DLL 函数的动态链接。

2）显式链接

显式链接就是指应用程序在执行过程中，程序自己通过专门的函数调用来动态加载 DLL。显式链接仅发生在应用程序的运行阶段，应用程序使用 LoadLibrary 加载 DLL，然后通过行函数调用以显式加载和卸载此 DLL，接下来显式定位 DLL 中的输出函数并调用，最后要通过函数调用卸载 DLL。显示链接需要用到的函数将在下节介绍。

通常，显式链接需要用到如下 3 个函数。

① LoadLibrary()函数

该函数用于加载 DLL 和获取模块句柄。该函数的原型是：

```
HMODULE LoadLibrary(LPCTSTR lpFileName  //DLL文件
);
```

lpFileName 为要加载的 DLL 的名字，其中可以包含完整的路径消息。

② GetProcAddress()函数

该函数用于获得指定动态链接库中的某个导出函数的指针。该函数的原型是：

```
FARPROC GetProcAddress(HMODULE hModule,  //DLL模块句柄
                      LPCTSTR lpProcName  //输出函数名或序号
);
```

hModule 是 HMODULE 类型的参数，它用于接收一个指向某个 DLL 的句柄，此句柄通过 LoadLibrary()函数获取；参数 lpProcName 是一个字符串，用来指明此函数调用需要获取的 DLL 中的哪个导出函数的地址。

③ FreeLibrary()函数

该函数将卸载动态链接库，并释放动态链接库模块所占用的内存空间。该函数的原型是：

```
BOOL FreeLibrary(HMODULE hLibModule //DLL模块句柄
);
```

参数 hLibModule 就是指向通过 LoadLibrary()加载的 DLL 的句柄。

3．动态链接库的创建和生成

下面我们重点用实例来创建和生成一个 MFC 常规动态链接库。

1）用“MFC DLL”向导创建与 MFC 动态链接的常规 DLL

STEP 01 打开 Visual C++ 2005，选择“文件”→“新建”→“项目”命令，在弹出的对话框中选择“MFC”选项，在“模板”列表中选择“MFC DLL”选项，输入要生成的 DLL 的文件名“MyProjectDLL”。在“位置”文本框中输入或选择要保存的路径，如图 2-36 所示。

STEP 02 单击“确定”按钮，进入“MFC DLL 向导”对话框，如图 2-37 所示。在“DLL 类型”中，选择“使用共享 MFC DLL 的规则 DLL”选项，其他可采用默认值。

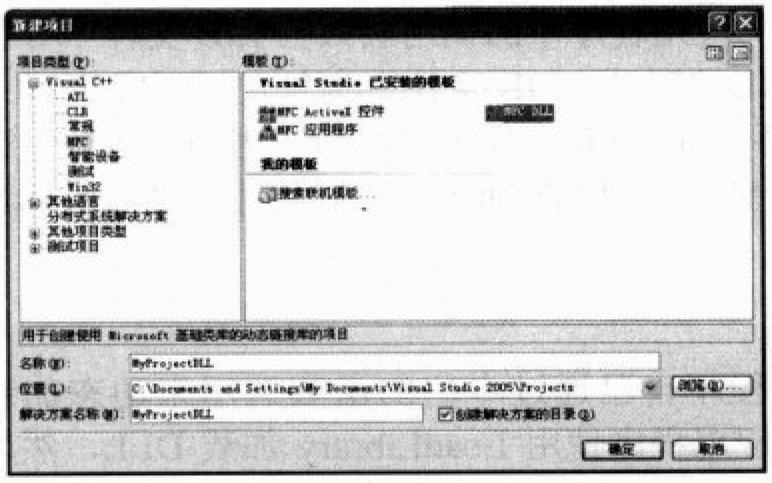


图 2-36　MFC 常规动态链接库新建项目

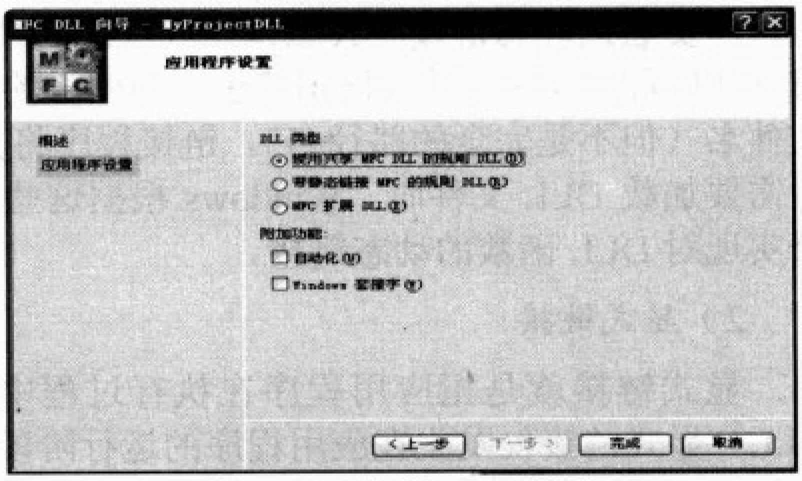


图 2-37　选择 DLL 类型

STEP 03 此时 MFC DLL 向导可自动完成动态链接库的创建，而在指定的目录下生成 DLL 项目所必需的全部文件，如图 2-38 所示。生成的动态链接库界面如图 2-39 所示。

图 2-38　解决方案资源管理器

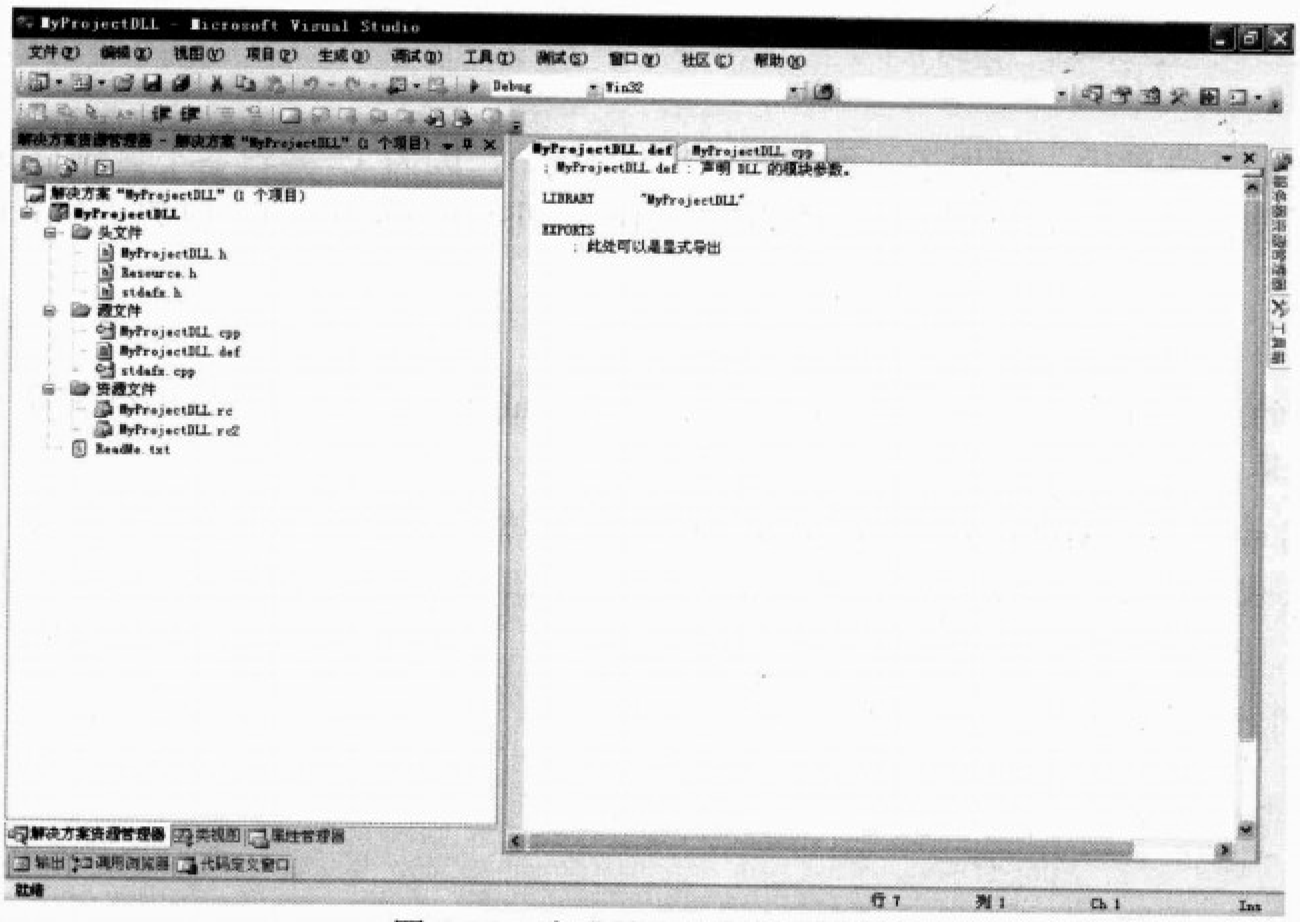


图 2-39　生成的动态链接库界面

2）定义动态链接库的输出函数

动态链接库的输出函数常常在.def 文件中定义，打开 MyProjectDLL.def 文件，看到 EXPORTS 语句下面还是空的，没有添加任何输出函数，然后把所有需要输出的函数添加到 MyProjectDLL.def 文件的 EXPORTS 语句后面，完成输出函数的定义。如图 2-39 所示。

在 DLL 中定义的全局变量，如果被输出，则各个进程都会获得该变量的一个复制变量。用_ declspec(dllexport)来输出全局变量，并且全局数据可以被初始化，例如：

```
_declspec(dllexport) int Var=0;
```

下面是 MyProjectDLL.h 头文件的定义，其中加入函数 Factorial (int a)来作为输出函数。

```
//MyProjectDLL.h : MyProjectDLLDLL的主头文件
#pragma once
#ifndef __AFXWIN_H__
#error "在包含此文件之前包含“stdafx.h”以生成PCH 文件"
#endif
#include "resource.h"          //主符号
extern "c"_declspec(dllexport) Factorial (int a);
//CMyProjectDLLApp
//有关此类实现的信息，请参阅MyProjectDLL.cpp
   class CMyProjectDLLApp : public CWinApp
{
public:
CMyProjectDLLApp();
//重写
public:
virtual BOOL InitInstance();
```

```
    DECLARE_MESSAGE_MAP()
};
```

在创建的头文件中，添加了一行代码（黑体所示），因为 C 编辑器不会进行这样的命名修饰，为了使得输出变量可以在 C 中使用，就在_declspec(dllexport)之前加上 **extern "c"**。

注意：

黑体的代码的最后我们加入了输出函数 Factorial(int a)，这个函数的作用是实现简单的阶乘运算。

下面编写源文件 MyProjectDLL.cpp，也就是编写输出函数 Factorial(int a)，如下所示。

```
extern "c" long _declspec(dllexport) Factorial(int a)
{
int i,out=1;
if(a==0)
    return 1;
else if(a<0)return 0;
else
      for(i=1;i<=a;i++) out=out*i;
   return out;
}
```

编译生成动态链接库文件 MyProjectDLL.dll，该文件在项目所在文件夹的 Debug 目录下。

2.3 利用向导完成基本开发

由于传统的 Windows 程序开发需要开发人员进行手工输入程序代码，那么，有没有更快捷的生成应用程序的方法呢？Visual C++ 2005 提供了各种向导和工具帮助开发人员实现所需的功能，这在一定程度上实现了软件的自动生成和可视化编程。利用应用程序向导工具，我们可以在很短的时间内创建一个 Windows 应用程序。应用程序在创建新项目时提供应用程序框架，自动生成框架程序的源文件，这样用户不必从头开始编写代码就能很快地构建出程序原型。

2.3.1 生成应用程序

应用程序向导可生成单文档界面（SDI）、多文档界面（MDI）和基于对话框的应用程序。

STEP 01 打开 Visual C++ 2005，选择“文件”→“新建”→“项目”命令，在弹出的对话框中选择“Visual C++”中的“MFC”选项，接着选择“MFC 应用程序”选项，表示要创建一个 Windows 应用程序，具体如图 2-40 所示。在下方的文本框中输入项目的名称和项目要保存的位置，而“解决方案名称”在默认情况下与项目名称相同。单击“确定”按钮，进入下一步。

STEP 02 MFC 应用程序向导的第一个界面是“欢迎使用 MFC 应用程序向导”界面，它

概述了当前的项目设置。如图 2-41 所示。单击“下一步”按钮，进入下一步。

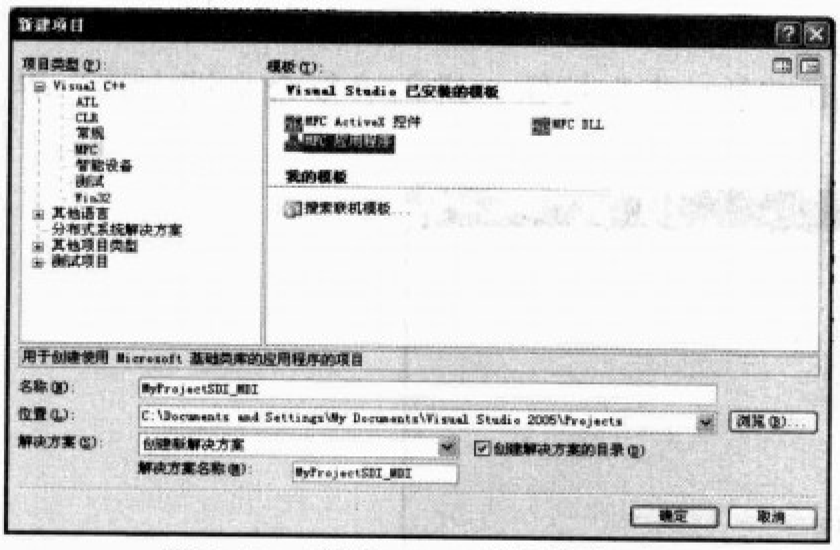


图 2-40　创建 MFC 应用程序项目

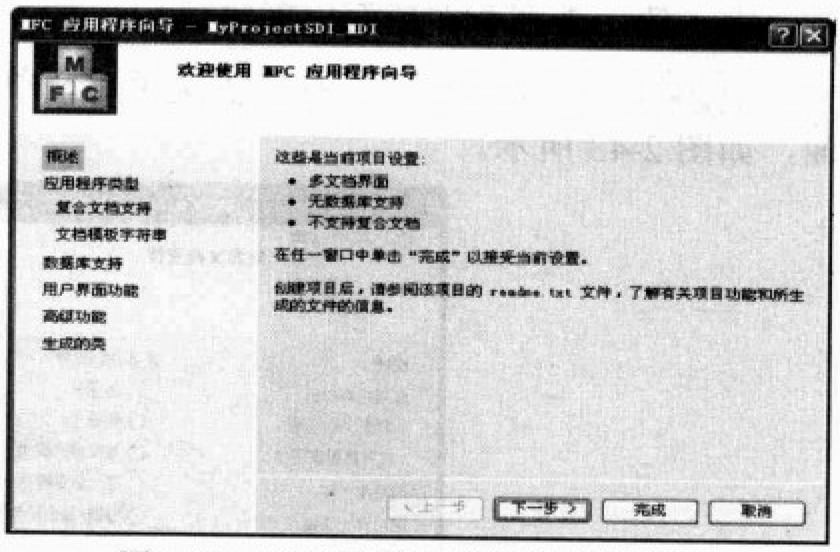


图 2-41　MFC 应用程序向导欢迎界面

STEP 03 此时弹出了“应用程序类型”界面，其中在“应用程序类型”、“项目类型”等选项中，用户可根据需要选择样式类型，从而生成不同的应用程序代码，如图 2-42 所示。

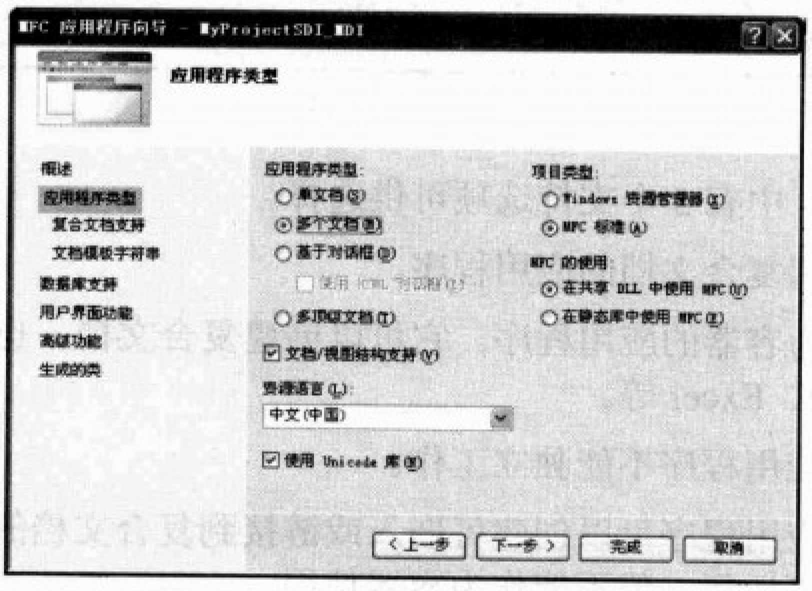


图 2-42　选择应用程序类型界面

主要有 4 种应用程序界面样式供用户选择。

- 单文档：单文档界面的应用程序（SDI）。单文档一次只能打开一个文档。典型的有 Windows 自带的写字板、记事本等。
- 多文档：多文档界面的应用程序（MDI）。它每次可以同时打开多个文档或者是多个视图。
- 基于对话框：基于对话框的应用程序。
- 多顶级文档：它和多文档应用程序类似，当打开的子文档不在父文档容器内时，和父文档有一样的样式。

这 4 种样式可供用户选择，当用户选择哪种样式时，左上角的界面类型示意图就会显示用户的选择样式。

“文档/视图结构支持”复选框表示生成的应用程序是否支持“文档/视图”结构。在通常情况下，对于初学者，最好选中，否则应用程序框架不支持文档/视图结构，也不能生成文档类。

“资源语言”下拉列表框提供了支持应用程序的语言。用户最好确保此处所选语言与系统语言一致，否则会在后面出现错误。

STEP 05 根据需要，选择所需选项后，单击“下一步”按钮，进入“复合文档支持”界面，如图 2-43 所示。

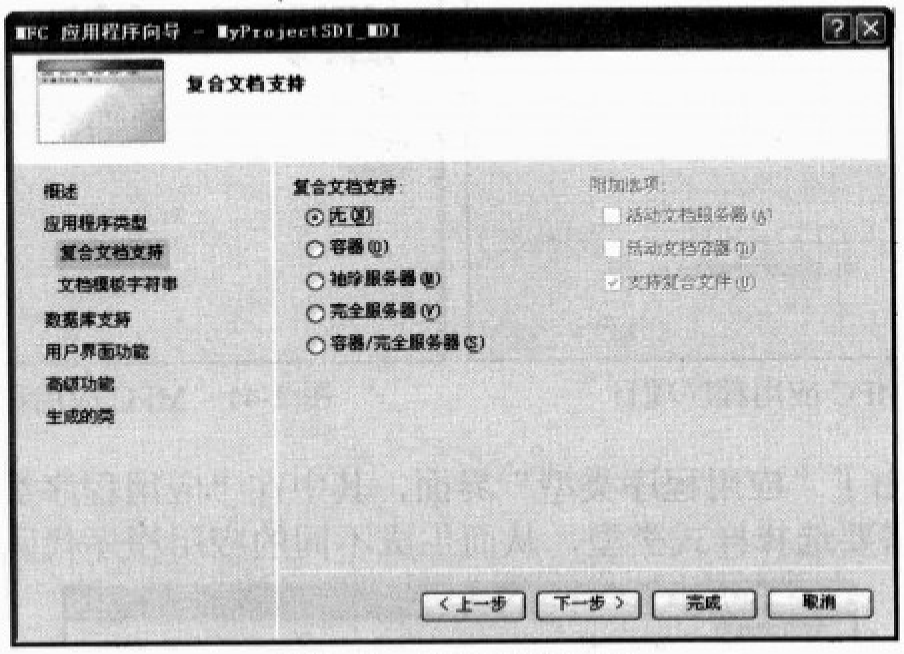


图 2-43　复合文档支持

在“复合文档支持”中有 5 个支持选项可供选择。

- 无：不编写支持复合文档的应用程序。
- 容器：创建作为容器的应用程序。它可以创建复合文档，也可以包含嵌入或链接的对象，如 Word、Excel 等。
- 袖珍服务器：应用程序不能独立工作。
- 完全服务器：应用程序可以创建可嵌入或链接到复合文档的对象，并且也能够作为单独的应用程序运行，但不能作为容器使用。
- 容器/完全服务器：应用程序既可以是容器也可以是服务器。

STEP 06 此时选择“无”单选按钮，单击“下一步”按钮，进入“文档模板字符串”界面，如图 2-44 所示。

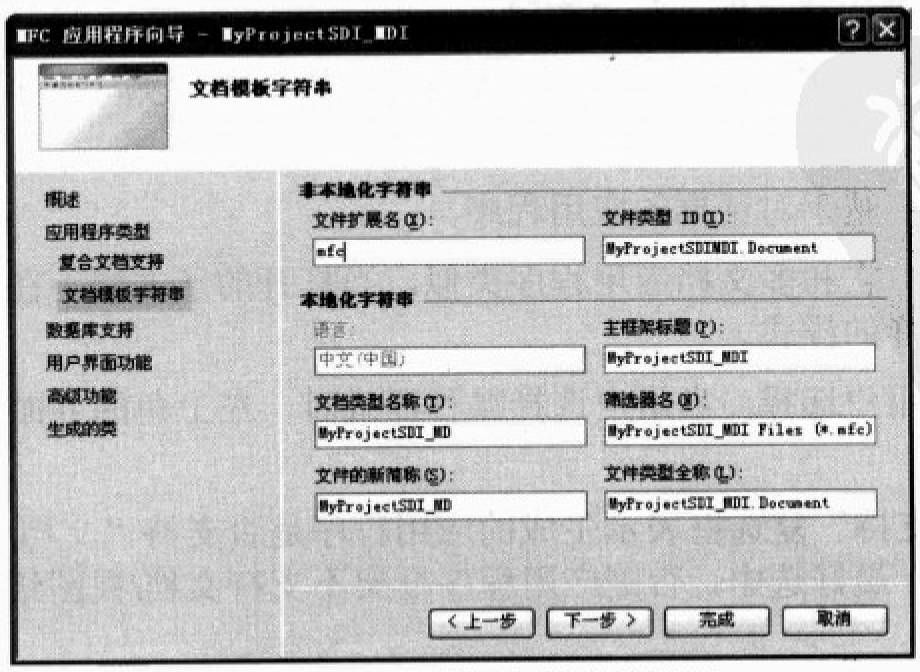


图 2-44　文档模板字符串的设置

“文档模板字符串”中的几个选项用来设置文档参数，各项意义如下。

- 文件扩展名：即应用程序创建文档的后缀名。
- 文件类型 ID：应用程序创建的文件类型。
- 主框架标题：即所创建的项目名称。
- 文档类型名称：即创建的文件名称。
- 文件的新简称：不带文件后缀名的文件名称。
- 筛选器名：系统自动生成。
- 文件类型全称：即所创建的文件类型。

按照上图所示采用设定值，在“文件扩展名”文本框中输入“.mfc”，则每次打开文件或保存文件时，都会显示“MyProjectSDI_MDI(*.mfc)”。

STEP 07 单击“下一步”按钮，进入“数据库支持”界面，如图 2-45 所示。

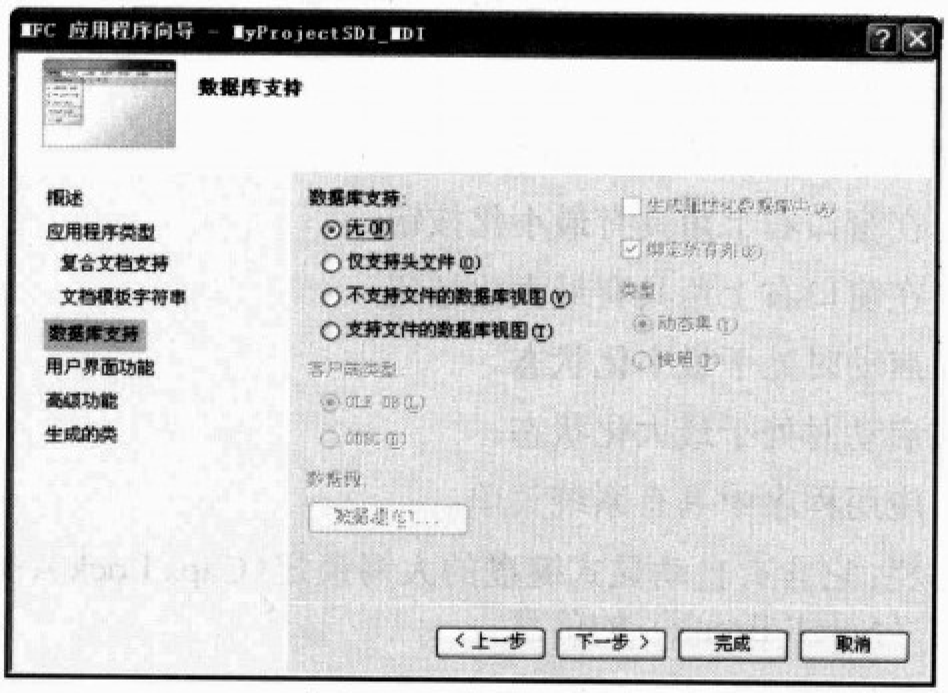


图 2-45　数据库支持

此对话框中有以下 4 个选项。

- 无：不支持数据库。
- 仅支持头文件：访问数据库，不能从 CFormView 类派生视图类，或者其界面不含记录菜单。
- 不支持文件的数据库视图：在访问数据库时，可以从 CFormView 类派生视图类，并且包含记录菜单，可以更新数据库，但无法保存文档。
- 支持文件的数据库视图：在访问数据库时，可以从 CFormView 类派生视图类，并且包含记录菜单，可以更新数据库，可以保存文档。

STEP 08 大部分应用程序不需要数据库支持，此刻选择“无”单选按钮。选择完成后，单击“下一步”按钮，进入“用户界面功能”界面，如图 2-46 所示。

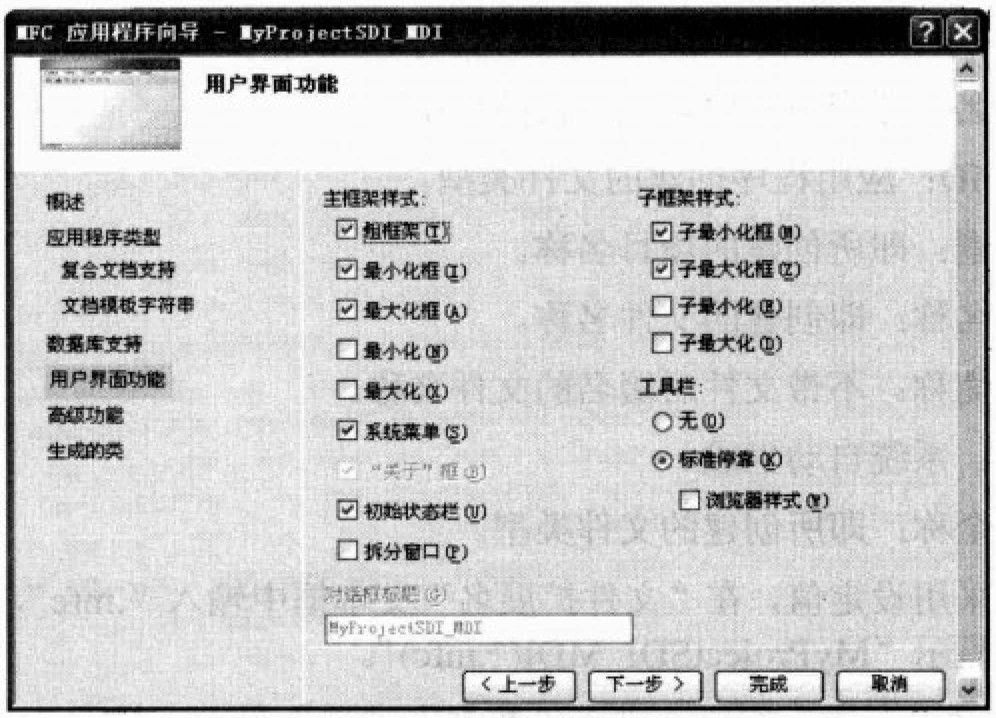


图 2-46　用户界面功能

此对话框中的各选项意义如下。

- 粗框架：窗口可以有厚边并且能够调节大小。
- 最小化框：在窗口右上角具有最小化按钮。
- 最大化框：在窗口右上角具有最大化按钮。
- 最小化：在启动时处于最小化状态。
- 最大化：在启动时处于最大化状态。
- 系统菜单：应用程序中具有系统菜单。
- 初始化状态栏：它含有自动显式键盘的大写锁定（Caps Lock）、数字锁定（Num Lock）和滚动锁定（Scroll Break）的信息。
- 拆分窗口：可以添加拆分窗口实现。
- 子最小化框：在多文档（MDI）子框架窗口右上角有最小化按钮。
- 子最大化框：在多文档（MDI）子框架窗口右上角有最大化按钮。
- 子最小化：在多文档（MDI）子框架窗口启动时处于最小化状态。
- 子最大化：在多文档（MDI）子框架窗口启动时处于最大化状态。
- 浏览器样式：可以在工具栏上添加任何 Windows 控件，且工具栏有浮动的效果。

STEP 09 选择完成后，单击“下一步”按钮，进入“高级功能”界面，如图 2-47 所示。各个选项的意义如下。

- 区分上下文的帮助：为应用程序在“帮助”菜单中添加“索引”和“使用帮助”菜单项，并生成这些功能的代码。
- 打印和打印预览：为应用程序在“文件”菜单中添加“打印”和“打印预览”菜单项，并生成这些功能的代码。
- 自动化：允许应用程序向脚本工具和其他应用程序公开对象。
- ActiveX 控件：使应用程序包含 Active 控件。

- MAPI（消息处理 API）：仅当用户具有电子邮件系统时有效。
- Windows 套接字：是否支持 Windows 套接字进行基于 TCP/IP 的网络通信。
- Active Accessibility：看应用程序是否支持 Active Accessibility。
- 公共控件清单：生成应用程序清单，此清单启用 Windows XP 一起提供的新的公共控件。

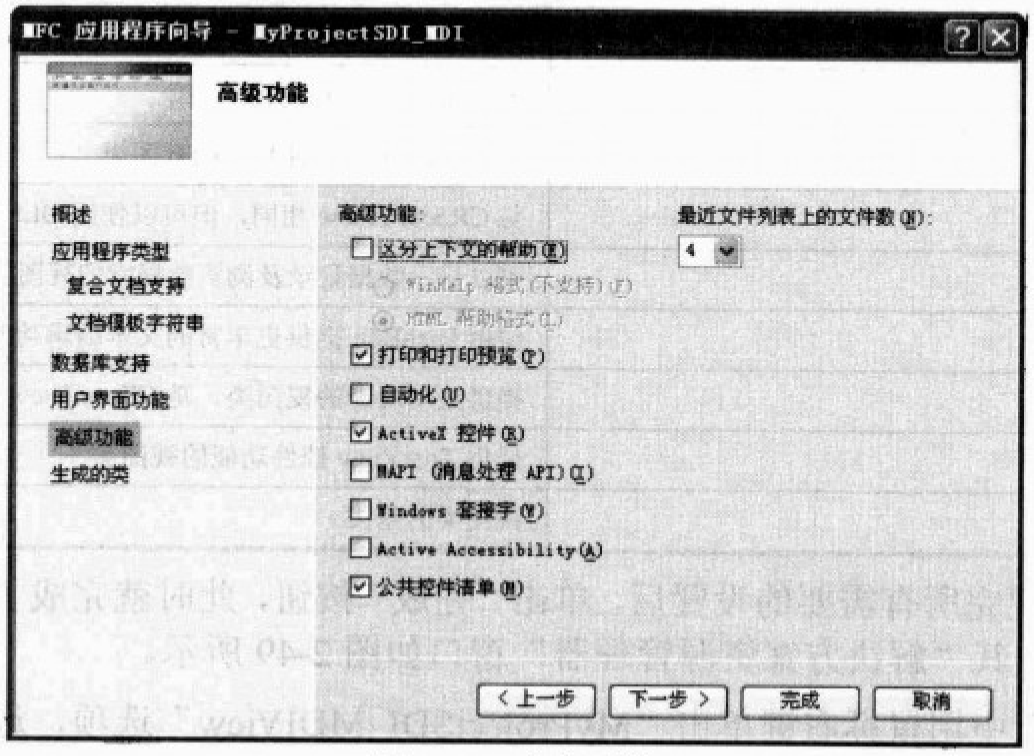


图 2-47　高级功能

STEP 10 设置完成后，单击“下一步”按钮，进入“生成的类”界面，如图 2-48 所示。

这是利用向导生成应用程序的最后一步，主要是确定自动生成的类名和文件名，一般情况下，无须改变这些名字，表 2-1 列出了不同的视图类。

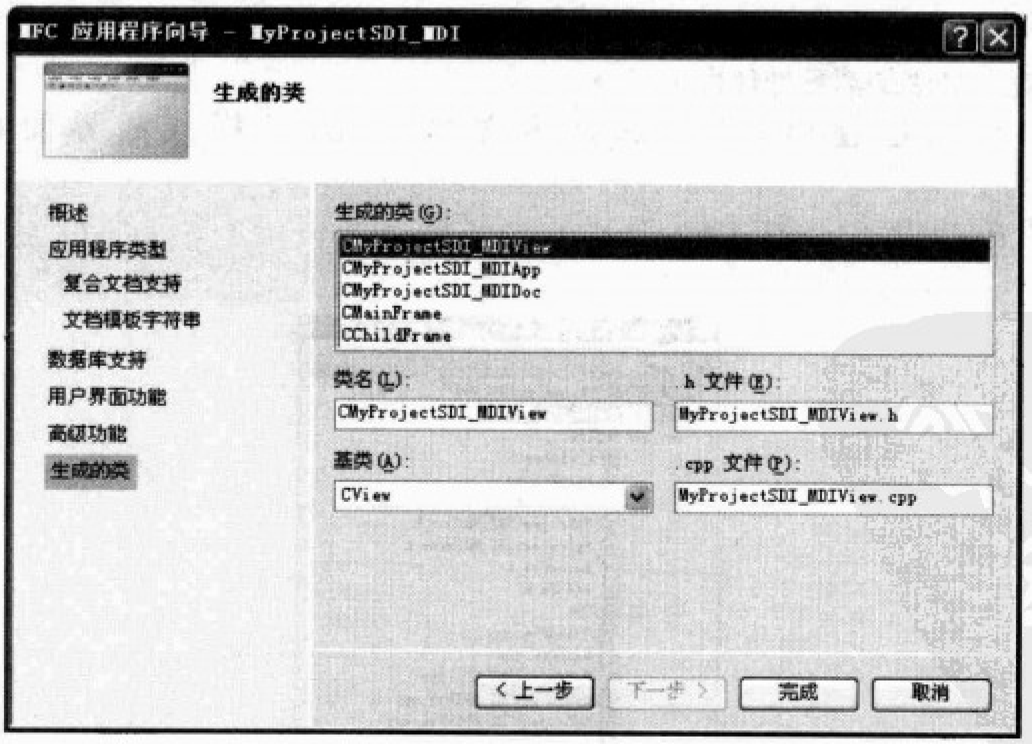


图 2-48　生成的类

表 2-1　常用的视图类

视图类	说明
CCtrlView	实现 Windows 常用控件的视图类
CDAORecordView	与 CRecordView 相同，但可以使用 DAO 数据库类
CEditView	提供基本文本编辑功能的视图类
CFormView	提供对话框资源实现窗体式视图
CHtmlView	可以显示 HTML 的视图类
CListView	提供 ListView 控件功能的视图类
COleDBRecordView	与 CRecordView 相同，但可以使用 OLE DB 数据库类
CRecordView	可以显示数据记录及浏览数据库的视图类
CRichEditView	使用 RichEdit 提供更丰富的文本编辑功能的视图类
CScrollView	提供滚动功能的视图类，是 CFormView 的基类
CTreeView	提供 TreeView 控件功能的视图类
CView	视图类的基类

STEP 11 设置完所有需要的设置后，单击“完成”按钮，此时就完成了应用程序的生成。在生成的界面中，其“解决方案资源管理器”窗口如图 2-49 所示。

在“类视图”中用鼠标右键单击“MyProjectSDI_MDIView”选项，选择“属性”，即可打开“MyProjectSDI_MDIView”属性面板，单击“消息”图标，如图 2-50 所示。在列表中选择“WM_PAINT”，如图 2-51 所示。选择“<添加>OnPaint”，出现代码编辑区。

```
//CMyProjectSDI_MDIView 消息处理程序
void CMyProjectSDI_MDIView::OnPaint()
{
CPaintDC dc(this); //device context for painting
   //TODO: 在此处添加消息处理程序代码
dc.DrawText(L"欢迎使用MFC，这是单文档（SDI）样式的界面！",CRect(50,50,
500,100),DT_LEFT);
   //不为绘图消息调用CView::OnPaint()
}
```

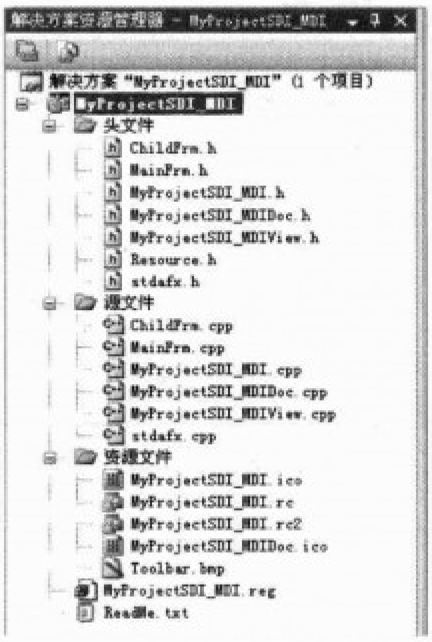


图 2-49　解决方案资源管理器

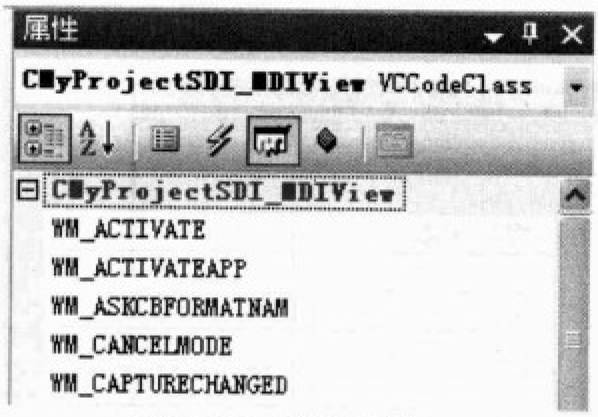


图 2-50　消息属性

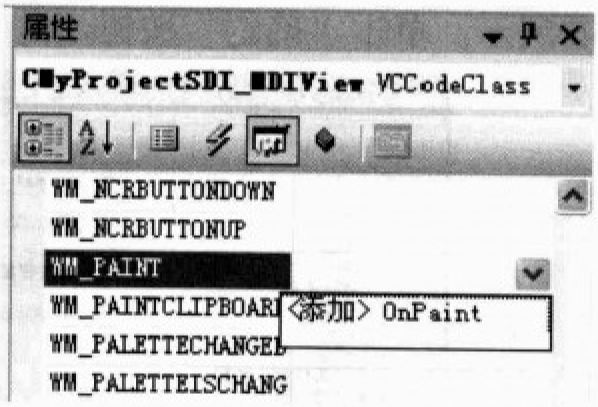


图 2-51　增加消息响应函数

STEP 12 编译（Ctrl+F7）、运行（Ctrl+F5），此界面是多文档（MDI）样式，如图 2-52 所示。图 2-53 所示是单文档（SDI）样式界面。

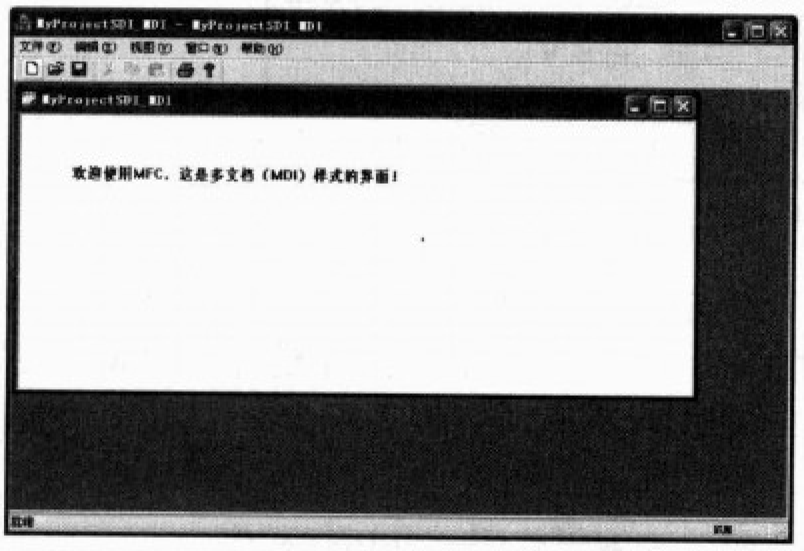


图 2-52　MDI 运行结果

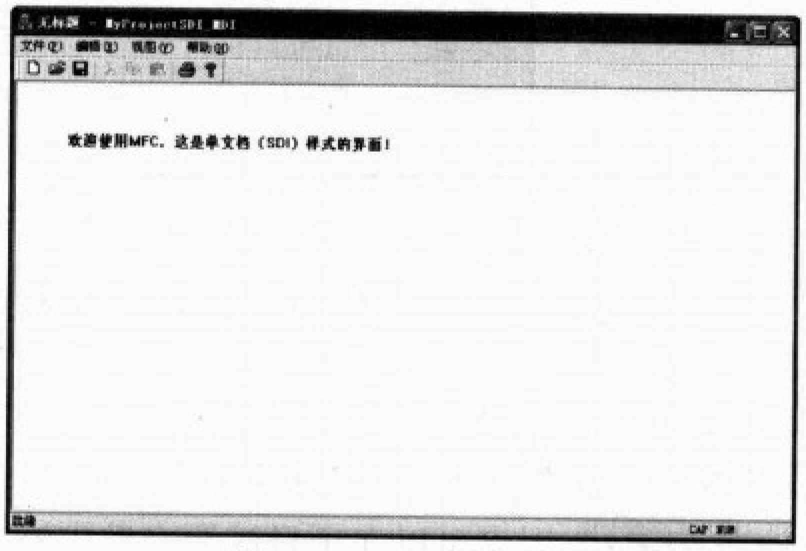


图 2-53　SDI 运行结果

STEP 13 当用户选择运行结果界面的“帮助”→“关于 MyProjectSDI_MDI…”命令，即可看到一个“关于 MyProjectSDI_MDI”对话框，如图 2-54 所示。

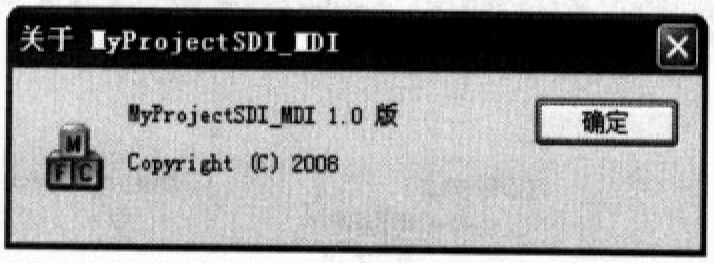


图 2-54　“关于 MyProjectSDI_MDI”对话框

2.3.2　生成应用库LIB/DLL

生成应用库 LIB 与生成应用库 DLL 的方法类似，下面通过实际的步骤来讲解这两种类的生成方式。

1．生成 LIB 应用库

STEP 01 打开 Visual C++ 2005，选择“文件”→“新建”→“项目”命令，弹出“新建项目”对话框，选择左侧面板“Visual C++”中的“Win32”选项，在右侧“模板”中选择“Win32 控制台应用程序”，然后分别在“名称”、“位置”文本框中输入项目名称和要保存的位置，

如图 2-55 所示。单击“确定”按钮，进入下一步。

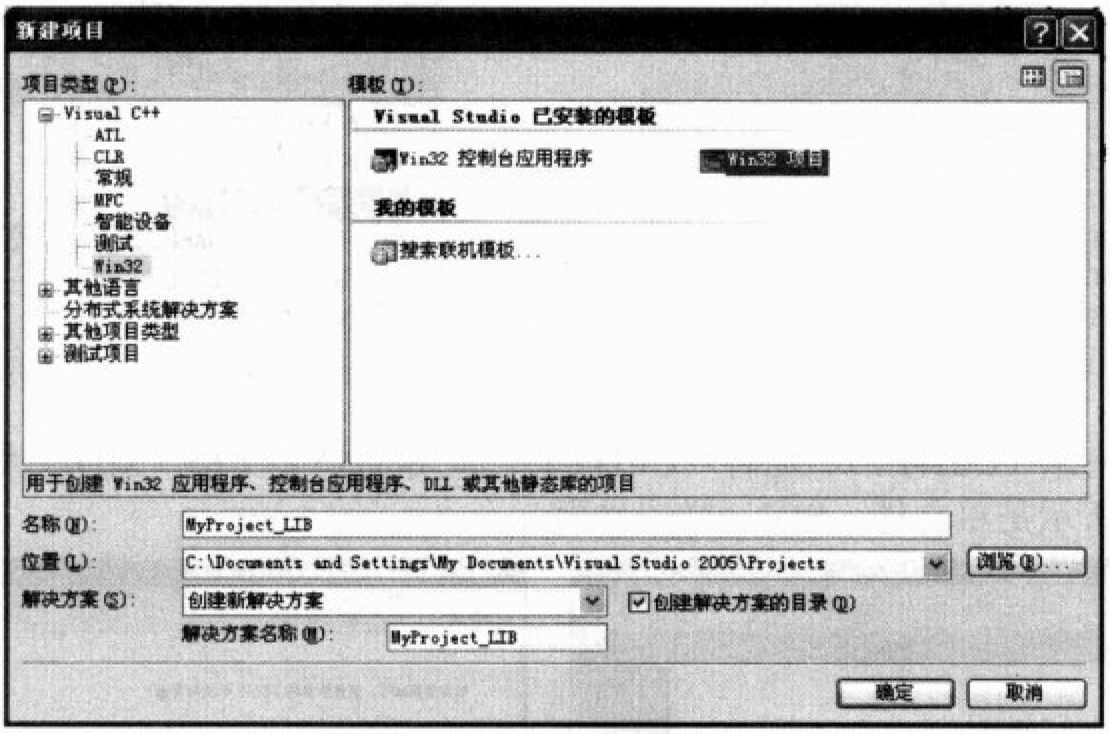


图 2-55　新建项目的设置

STEP 02 在弹出的“Win32 应用程序向导”对话框中，选择左侧的“应用程序设置”选项，再在右侧“应用程序类型”选项中选择“静态库”，在“附加选项”中选择“预编译头”，如图 2-56 所示。单击“完成”按钮，进入所生成的用户界面，如图 2-57 所示。其中“解决方案资源管理器”窗口如图 2-58 所示。

STEP 03 用户可以在项目的源文件中添加代码，来编辑代码，并运行程序。

至此，我们简单介绍了如何用 Visual C++ 2005 的应用程序向导生成一个简单的静态链接库。

图 2-56　选择静态库

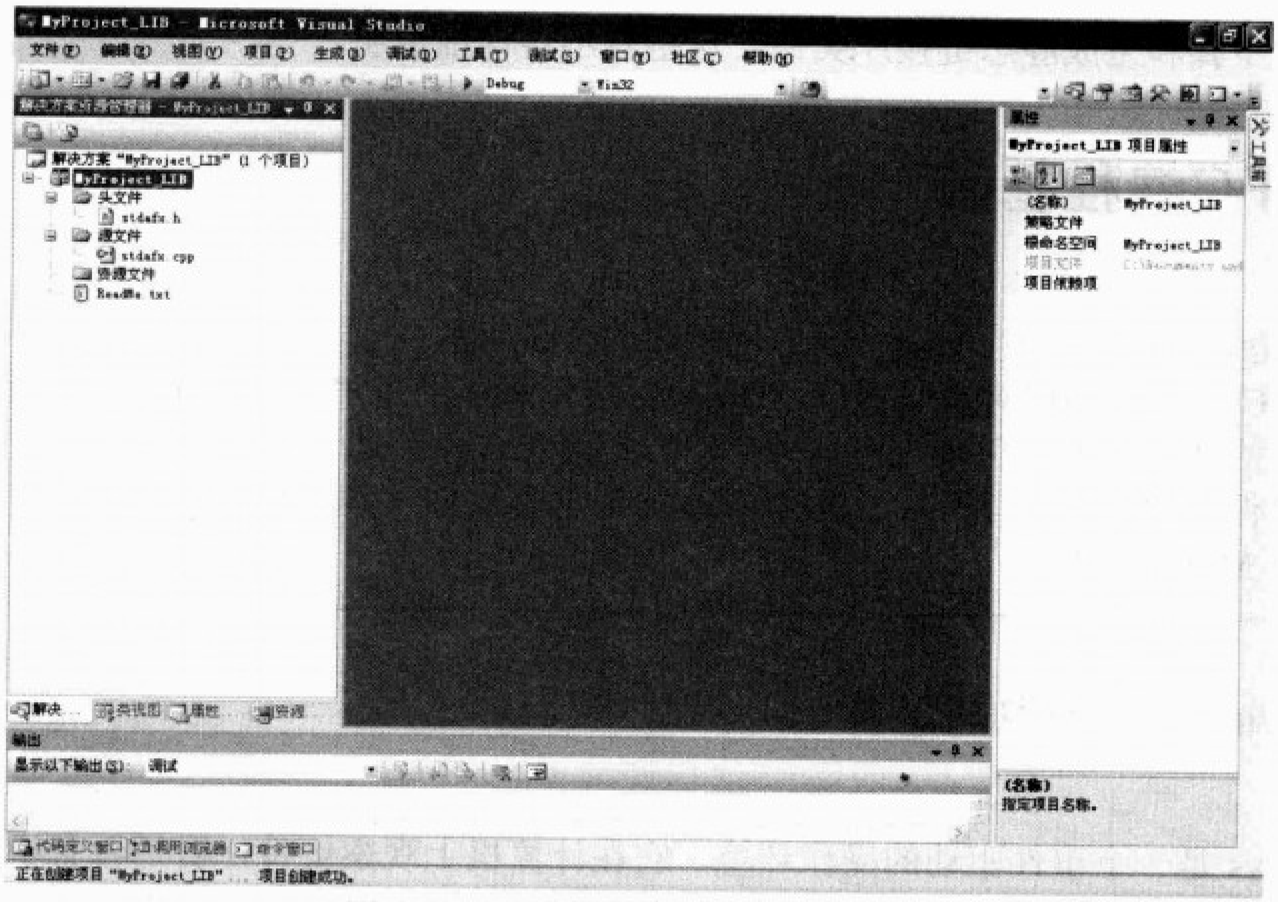


图 2-57　生成的静态链接库项目界面

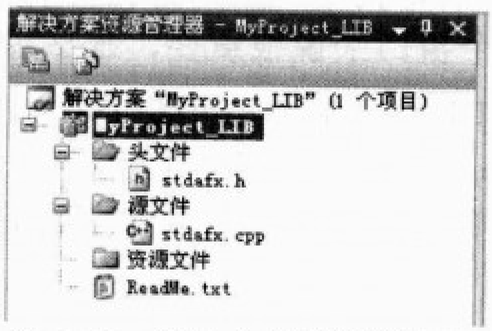


图 2-58　解决方案资源管理器

2．生成 DLL 应用库

利用向导生成动态链接库 DLL，与生成静态链接库 LIB 有一点不同，就是在 Win32 应用程序向导所生成的“应用程序设置”界面中选择“DLL”单选按钮，如图 2-59 所示。

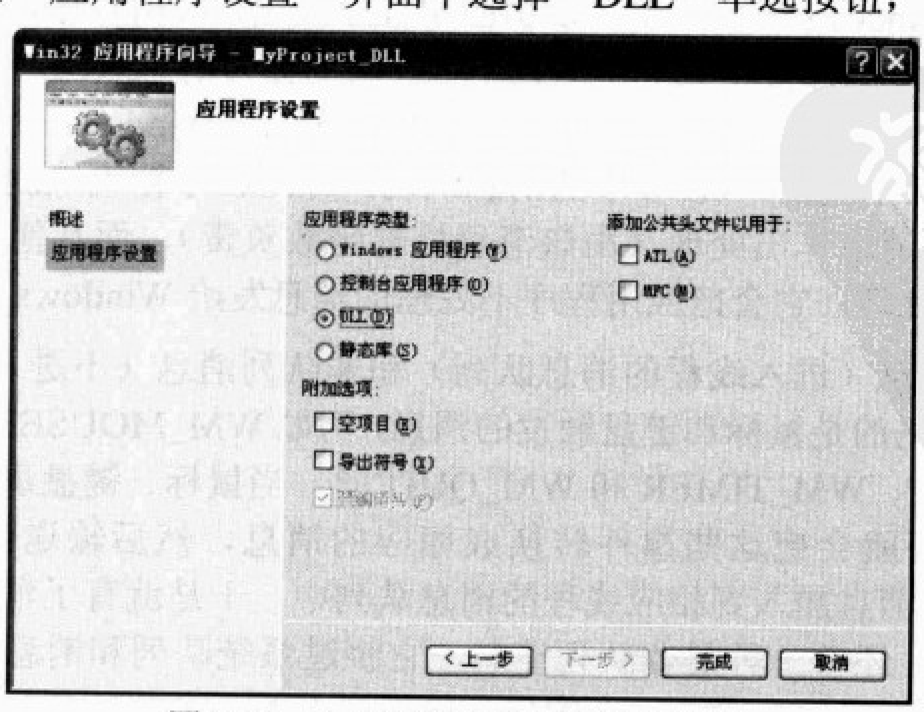


图 2-59　在应用程序设置中选择 DLL

其他的步骤和生成静态链接库类似，这里就不再详细说明。

2.4 MFC编程基础

MFC 是编程人员开发 Windows 应用程序的必备工具，它为应用程序的开发提供了强大的支持。MFC 主要的编程对象是窗口，Windows 应用程序通过消息系统来实现图形用户界面。系统把用户的任何动作都包装在消息中，然后把消息传递给适合的窗口过程处理，这样就实现了基于事件的驱动。只有深入了解 Windows 消息和事件及其消息的循环和事件的响应，才有可能精通 MFC 编程。

2.4.1 Windows消息与事件

Windows 是一个事件驱动的操作环境，它在计算机上观察硬件，看是否有事情发生，一旦发生了一些有趣的事，例如鼠标移动了，Windows 就会检测到它，并将消息传递给与消息对应的应用程序。这个关于事件的信息就是一条消息。上面的例子用到了 WM_MOUSEMOVE 消息。如果这个消息对屏幕上某个窗口来说很重要，那么窗口就会处理这个消息。

所谓消息（Message）就是用于描述某个事件发生的信息，而事件（Event）则是用户操作应用程序产生的动作或 Windows 系统自身所产生的动作。事件和消息两者密切相关，事件是原因，消息是结果，事件产生消息，消息对应事件。如上面所说，用户按下键盘或移动鼠标就标志发生键盘事件或鼠标事件，系统就会产生一条对应的键盘消息或鼠标消息。

在 Windows 这个多任务多进程的操作系统中，多个应用程序之所以能够有序地运行，就在于它的事件驱动、消息响应机制。所有的外部事件（如鼠标操作、键盘操作和定时器事件等）都被 Windows 拦截，转换成消息后再发送到应用程序中的目标对象，应用程序对消息作出响应并进行相应的处理。消息处理机制是 Windows 编程的灵魂，Windows 程序利用消息与其他 Windows 程序或 Windows 系统进行通信。

消息不仅可由 Windows 发出，也可以由应用程序发出，应用程序还可以自定义消息。发送到每个程序窗口的消息都会排成一个队列，这种队列称为消息队列。如图 2-60 所示，每一个应用程序都维护着自己的消息队列，当某种事件发生时，Windows 会判断当前事件属于哪个程序，进而将对应的消息加入该程序的消息队列，应用程序轮流检测消息队列中的消息，但只对需要它处理的消息作出处理（由该消息处理函数负责），而其他消息有默认的窗口函数 DefWindowProc()处理（它会把应用程序不处理的消息发给 Windows 操作系统）。

消息分为队列消息（进入线程的消息队列）和非队列消息（不进入线程的消息队列）。对于队列消息，最常见的是鼠标和键盘触发的消息，例如 WM_MOUSERMOVE、WM_CHAR 等，以及 WM_PAINT、WM_TIMER 和 WM_QUIT 等。当鼠标、键盘事件被触发后，相应的鼠标或键盘驱动程序就会把这些事件转换成相应的消息，然后输送到系统消息队列，由 Windows 系统负责把消息加入到相应线程的消息队列中，于是就有了消息循环（从消息队列中读取并派送消息）。还有一种是非队列消息，它绕过系统队列和消息队列，直接将消息发送到窗口过程。例如，当用户激活一个窗口系统发送 WM_ACTIVATE、WM_SETFOCUS 和

WM_SETCURSOR 创建窗口时发送 WM_CREATE 消息。

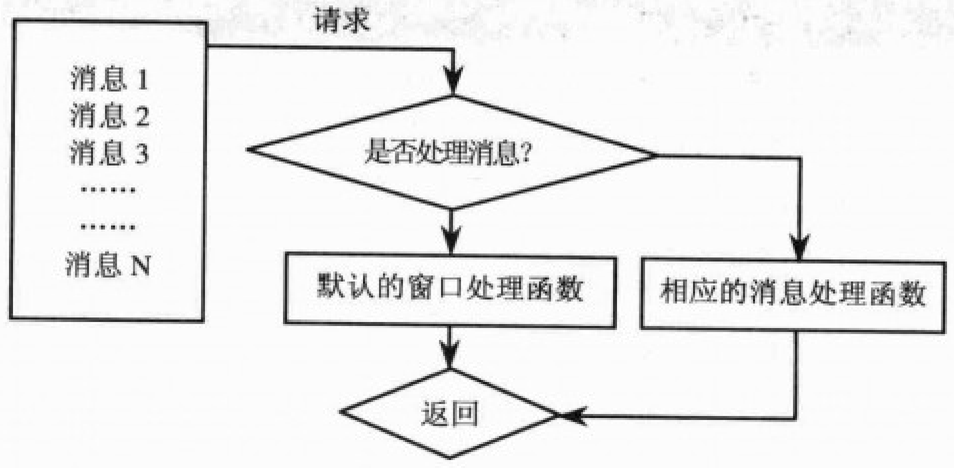


图 2-60　消息的处理机制

Windows 操作系统中的消息可以分为两大类:系统定义的消息和应用程序自定义的消息。下面将首先介绍这两类消息，然后介绍有关消息和事件的响应和处理。

1．系统定义的消息

操作系统可以向应用程序发送或投递系统定义的消息控制应用程序的行为并提供输入。例如，对每一个输入事件，用户按键、移动鼠标、单击控件都会产生一条消息。应用程序通过响应这些消息处理用户的动作，实现和用户的交互。应用程序也可以向系统发送或投递系统定义的消息以控制预先注册的类型的控件窗口的行为。

每一个系统定义的消息都有唯一的标识符常量。在 SDK（Soft Development Kit，软件开发包）头文件中这些常量被定义为 32 位整数，标识符常量的字符分为两部分，前缀和后缀。前缀表示处理该消息的窗口的类别，后缀描述了该消息的目的。例如，WM_PAINT 的前缀 WM 表示负责处理该消息的窗口为一般窗口，后缀 PAINT 表示请求窗口绘制其内容。表 2-2 列出了系统定义消息的全部前缀和说明。

表 2-2　消息的全部前缀和说明

前缀	说明
BM	按钮（Button Control）
CB	组合框（Combo Box Control）
CDM	通用对话框（Common Dialog Box）
DBT	设备消息（Device Control）
DL	下拉列表（Drag List Box）
EM	编辑框（Edit Control）
HKM	热键（Hot Key Control）
IPM	IP 控件（IP Address Control）
LB	列表框（List Box Control）
LVM	列表视图（List View Control）

（续表）

前缀	说明 .
MCM	日历控件（Month Calendar Control）
PBM	进度条（Progress Bar）
PSM	属性（Property Sheet）
RB	伸缩条（Rebar Control）
SB	状态栏（Status Bar Window）
SBM	滚动条（Scroll Bar Control）
STM	静态条（Static Control）
TB	工具条（Toolbar）
TBM	跟踪条（Trackbar）
TCM	标签控件 Tab Control
TVM	树视图（Tree-view Control）
UDM	微调按钮控件（Up-down Control）
WM	普通窗口（General Window）

Windows 中大概有 400 多种消息，这些消息可以进一步细分为 3 类。

- 窗口消息（Windows Message）：也叫 Windows 消息。是由操作系统和控制其他窗口的窗口所使用的消息。所有的以“WM_”为前缀的消息都是 Windows 消息，这一类消息有 WM_CREATE、WM_DESTROY 和 WM_MOVE 等。这类消息有窗口和视图（文档/视图模型中的视图）处理，并且通常含有用于指示如何对消息进行处理的函数。

任何由 CWnd 类派生的 MFC 库类都能处理一个标准的 Windows 消息；这意味着窗口消息可以对 CFrameWnd 对象、CMDIFrameWnd 对象、CMDIChildWnd 对象、Cview 对象、Cdialog 对象和由这些基类派生的任意类进行处理。

- 命令消息（Command Message）：是一种特殊的窗口消息，它从一个窗口被发送到另一个窗口以处理来自用户的请求。按钮窗口发送到父窗口的消息就是命令窗口。它通常来自与用户界面对象（包括菜单项、工具栏按钮、快捷键等）的 WM_COMMAND 消息。每个对象都有一个 ID，通过为对象和命令分配同一 ID 可以将界面对象和命令联系起来。
- 控件通知消息（Control Notification）：也可以叫控制通知或者控件通知。类似于命令消息，当用户与控件窗口交互时，这一类消息就从控件窗口发送到父窗口。但是，这种消息的目的并不在于处理用户命令，而是为了让父窗口能够改变控件，如修改控件外观、加载并显示更多的数据等。考虑按钮单击事件，可以设想按钮发送了鼠标单击消息给它的父窗口，这个消息也可以看成一个控件通知消息。

控件通知消息同窗口消息一样，可以由从 CWnd 类派生的任意的 MFC 类进行处理。但是控件通知消息同窗口消息有一个重要的区别：控件通知消息不会出现在消息映射中，而且用户也不需要使用向导将它们加入到应用程序中去。因此在向导对话框中，也不会发现任何

控件通知消息。

2. 应用程序自定义的消息

这种消息是由应用程序而不是在 SDK 头文件中定义的。应用程序利用这些消息来指示它所拥有的窗口完成特定的任务或同其他应用程序的窗口通信。如果应用程序创建了自己的消息，那么收到该消息的窗口过程必须解释该消息并提供合适的处理。

应用程序要为自定义的消息赋予一个唯一的 32 位标记。必须保证该标记和已有的系统定义的消息的标识符不冲突，有两种实现办法。

- 确保值在 0x0400（WM_USER）和 0x7FFF 之间。操作系统保留了 0x000 和 0x03FFF（WM_USER-1）之间的值，应用程序不能使用。
- 向 RegisterWindowMessage 传入一个消息字符串，获得一个介于 0xC00 和 0xFFFF 之间的系统范围内的唯一的消息标识符：

```
UNIT RegisterWindowMessage(
LPCTSTR lpString    //消息字符串
);
```

3. 消息队列

前面提到，窗口本身仅提供了处理消息的窗口过程，真正的处理过程是由创建窗口的线程来进行的。一个线程建立一个窗口后，操作系统会自动销毁线程拥有的所有窗口。建立线程必须是为窗口处理所有消息的线程。对一个线程，如果它至少建立了一个窗口，则应该由系统为它分配一个容纳待处理的消息的队列，这个队列主要用于窗口消息的派送，因此这个队列就是线程的消息队列。为了使窗口接收这些消息，线程必须要有自己的消息循环。

如何提供一个强大的应用程序运行环境是 Windows 设计的目标之一。所以每个线程必须有完全不受其他线程影响的消息队列，而且每个线程都必须有一个模拟环境，使线程可以维持它自己的键盘焦点、窗口激活、鼠标捕捉等。

为了降低线程对系统资源的要求，当一个线程第一次被建立时，系统假定线程不会被用于任何相关的任务，这样就不会创建消息队列，一旦这个线程调用一个与图形用户界面有关的函数，系统就会为该线程分配一个 THREADINFO 结构，并将这个数据结构与线程联系起来。THREADINFO 结构中包含一组成员变量，特别之处在于该结构中包含了该线程的消息队列集合，利用这组成员，线程可以认为它是在自己的独占的环境中运行。

为了更好地处理消息，一个线程将消息队列细分为若干类，如登记消息队列、虚拟输入队列、发送消息队列、应答消息队列等。因此，一个线程拥有不止一个消息队列。

线程通过消息循环从消息队列中取出消息，并分派到特定的窗口过程处理该函数。在处理消息的过程中可能又会产生其他消息，这些消息被放到消息队列中。如此不断往复，直到应用程序调用 PostQuitMessage 为止。

一般情况下，最新的消息总是放到队列的末尾。为了将消息投递到线程的消息队列，系统首先要依据 MSG 结构进行过滤，然后将 MSG 结构复制到消息队列。下面是 MSG 结构的定义。

```
Typedef struct tagMSG{HWND hwnd;
```

```
        UNIT message;
        WPARAM wParam;
        LPARAM lParam;
        DWORD time;
        POINT pt;
        };
```

在 MSG 结构中，前 4 个参数被称为**消息参数**。

- hWnd：表示处理该消息的窗口，该窗口的窗口过程将被调用以处理消息。
- message：消息的唯一标志。
- wParam：处理该消息所需要的参数。
- lParam：处理该消息的额外参数。

wParam 和 lParam 设定了处理该消息所需要的数据或数据的位置。其含义和值依赖于具体消息，依据消息的不同，它们可以包含一个整数、指向结构的指针等。如果消息不需要使用参数，则需要将 wParam 和 lParam 设为 NULL，窗口过程检查消息类型并根据消息类型决定如何解释 wParam 和 lParam。

前 4 个参数的含义在上面已详细介绍了。time 参数表示消息投递的时间，pt 参数表示鼠标位置。

线程可以通过调用 PostMessage 或者 PostThreadMessage 往自己或者其他线程的消息队列中投递消息。

应用程序通过调用 GetMessage 从消息队列中移出一条消息。如果仅仅是检查消息而不用移出，则可以调用 PeekMessage，该函数获得当前消息的一份复制。

注意：

在向窗口过程传递消息时，没有传递 time 和 pt 参数。应用程序在处理消息时，可通过调用 GetMessageTime 和 GetMessagePos 来获得这些消息。

并非所有的消息都会被放入消息队列中等待排队处理，还有一类消息直接发送到窗口过程进行处理。系统一般利用这种消息通知窗口某些事件发生了。

4. 消息循环

任何线程都拥有一组消息队列，并需要不断地从消息队列中取出消息进行处理，处理过程中又产生新的消息，如此往复构成消息循环。消息循环是 Windows 应用的典型结构。

在 Windows 系统中，每个线程都对应一个消息队列（Message Queue），这是系统自动创建的。所以，如果想让窗口程序响应鼠标键盘操作的话，就要从这个消息队列里取得消息，然后根据消息做相应的响应函数。一般的做法是写一个消息循环（Message Loop），不断地从消息队列里提取消息、处理消息，直到程序退出。MSDN 给出的例子如下。

```
MSG msg;
BOOL bRet;
while( (bRet = GetMessage( &msg, NULL, 0, 0 )) != 0)
{
```

```
    if (bRet == -1)
    {
        //错误处理
    }
     else
    {
TranslateMessage(&msg);       //产生一个新的消息投递到消息队列
DispatchMessage(&msg);        //调用目标窗口来处理消息
    }
}
```

其中的 GetMessage 就负责从消息队列中取得一个消息，它的原型如下。

```
GetMessage(LPMSG lpMsg,         //输出参数
HWND hWnd,  //窗口句柄
UINT wMsgFilterMin,             //第一个消息
UINT wMsgFilterMax              //最后一个消息
);
```

IpMsg 是指向 MSG 结构的指针，该结构从线程的消息队列里接收消息信息；hWnd 是取得消息窗口的句柄；wMsgFilterMin 是指定被检索的最小消息值的整数；WmsgFilterMax 是指定被检索的最大消息值的整数。

此函数返回–1、0 和其他值。–1 表示碰到错误；0 表示检索到了 WM_QUIT 消息，这时就要结束消息循环。

一般的情况是在主窗口的窗口过程中响应 WM_DESTROY 时，调用 PostQuitMessage 以结束消息循环。

```
Case WM_EESTROY:
//清理
PostQuitMessage(0);
Break;
```

线程的消息循环中必须包含对 TranslateMessage 的调用，以接收键盘输入。

```
BOOL TranslateMessage(CONST MSG* IpMsg   //消息信息);
```

每当用户按下键时，系统会产生两个虚拟键消息（WM_KEYDOWN 和 WM_KEYUP）。虚拟按键消息中包含了用户所按键的键代码，但没有包含键值。TranslateMessageeach 将把虚拟键消息翻译为字符消息（WM_CHAR），并把该消息投递到消息队列中。

注意：

TransalteMessageeach 并不修改 IpMsg 所指的消息，而是产生一个新的消息投递到消息队列中。

如果是为了其他目的而处理虚拟键消息，则应用程序不应该调用 TranslateMessage。例如，如果 TranslateAccelerator 返回非零值（这时意味着用户按下了某个组合键），那么此时就不应该调用 TranslateMessage，如下面的代码所示。

```
while(GetMessage(&msg,NULL,0,0))
{
```

```
        if(!TranslateAccelerator(msg.hwnd,hAccelTable,&msg))
        {
            TranslateMessage(&msg);     //接收键盘消息投递到消息队列
            DispatchMessage(&msg);      //向系统中顶层窗口发生消息
        }
    }
```

修改后的消息循环加入了对组合键的支持。TranslateAccelerator 判断待处理的消息是否是 WM_KEYDOWN 或者 WM_SYSKEYDOWN。若是，则进一步判断按键的虚拟码是否在 hAccelTable 所指的加速表中，如果在，则会产生 WM_COMMAND 或者 WM_SYSCOMMAND 消息并直接发送到合适的窗口过程中进行处理。

DispatchMessage 将调用目标窗口过程来处理消息。如果 MSG 的 hwnd 成员为 NULL，则 DispatchMessage 不会对该消息做任何处理；如果 hwnd 为 HWND_TOPMOST，则 DispatchMessage 向系统中所有顶层窗口发送消息。

即使一个线程创建了多个窗口，一个线程消息队列也仅需一个消息循环。这是因为 MSG 中的 hwnd 成员能指示 DispatchMessage 调用哪个窗口的窗口过程来处理该消息。

我们还可以用 PeekMessage 而不是 GetMessage 检索消息。PeekMessage 比 GetMessage 多了一个参数，通过该参数可以设定是否要从消息队列中移出消息。

```
    BOOL PeekMessage(LPMSG lpMsg,           //消息
                HWND hWnd,                  //窗口句柄
                UINT wMsgFilterMin,         //第一个消息
                UINT wMsgFilterMax,         //最后一个消息
                UINT wRemoveMsg             //是否移出该消息
                );
```

参数说明，lpMsg 是接收消息信息的 MSG 结构指针；hWnd 是其消息被检查的句柄；wMsgFilterMin 是指定被检查范围里的第一个消息；wMsgFilterMax 是指定被检查的消息范围里的最后一个消息；wRemoveMsg 确定消息如何被处理。

如果将 wPemoveMsg 设为 PM_NOREMOVE，则在读出消息后不会从消息队列中移出该消息；如为 PM_REMOVE，则读出消息后还会从消息队列中移出该消息。

下面介绍消息中一个很重要的概念，那就是句柄（Handle）。在 Windows 中我们常用句柄（Handle）标识程序中不同的对象或者同类中不同的实例。提供句柄可以访问各种资源，句柄是一个 4 字节的数，常用的句柄如表 2-3 所示。

表 2-3　常用句柄类型

句柄类型	说明	句柄类型	说明
HWND	窗口句柄	HFILE	文件句柄
HBRUSH	画刷句柄	HFONT	字体句柄
HBITMAP	位图句柄	HINSTANCE	实例句柄
HCURSOR	光标句柄	HMENU	菜单句柄
HCON	图标句柄	HPEN	画笔句柄
HDC	设备环境句柄	HPALETTE	调色板句柄

句柄并非对象的地址指针，它是作为系统内部表的索引值来使用的，这样就能避免应用程序直接同对象的内存地址打交道，体现了 Windows 资源管理的优越性。

句柄常作为 Windows 消息和 API 函数的参数，在采用 API 方法编写 Windows 应用程序时经常需要使用句柄。而采用 MFC 方法编写 Windows 应用程序，由于对应的 MFC 类已对句柄进行了封装，因此在多数情况下不再需要访问句柄。如果需要获取某个 MFC 类对象的句柄，则有两种方法。

- 提供访问类的一个 public 属性的成员变量，如 CWnd 类的成员变量 m_hWnd 就是一个窗口对象的句柄。
- 先定义一个句柄，然后调用 MFC 类的成员函数 Attch()将句柄与一个 MFC 类对象联系在一起，此时的句柄就成为 MFC 类对象的句柄。

2.4.2 消息与事件的响应

MFC 消息管理主要包括消息的发送和处理两个过程。消息发送，MFC 提供了类似于 API 函数功能的发送函数，而对于消息处理，MFC 则采用了消息映射机制，并利用类向导管理消息映射。MFC 消息处理的内部机制比较复杂，同时也是 MFC 编程中需要掌握的一个很重要的内容。

在 Windows 这个多任务多进程的操作系统中，多个应用程序之所以能够有序地运行，就在于它的事件驱动、消息映射机制（Message Map）。这种消息映射机制是通过一组消息映射宏（Macro）来实现的，其中一条消息映射宏把一个 Windows 消息与其消息处理函数链接起来，MFC 应用程序框架提供了消息映射功能，所有从 CCmdTarget 类派生出来的类都能够接收和处理消息，可以定义自己的消息映射表。其实消息映射机制就是一种消息分派机制。

而采用消息映射宏处理消息时，要将所有的消息映射集中在消息映射表中，在类的实现源文件中我们用宏定义消息映射表：BEGIN_MESSAGE_MAP()和 END_MESSAGE_ MAP()，例子如下。

```
BEGIN_MESSAGE_MAP(MyClass,BaseClass)
//…省略了消息映射宏
END_MESSAGE_MAP
```

MyClass 是消息映射的派生类名，BaseClass 是其基类名。消息映射宏通常以前缀 ON_开头，对于 MFC 预定义的消息，其后是消息名。和一般的宏一样，消息映射宏在编译期间被具体的源代码替换。

为了使用消息映射宏，需要在类定义的结尾用 DECLARE_MESSAGE_MAP()宏来声明使用消息映射宏，该宏对实现源文件中所定义的消息映射进行初始化。这通常在 MFC 应用程序向导中自动生成。

消息名决定了消息处理函数名，消息处理函数的命名规则是除去消息名的 WM_前缀，加上 ON 前缀。在声明处理函数名时，MFC 会在函数前面加上 AFX_MSG 标识。

Windows 应用程序中不同对象之间或不同程序之间都可以相互发送消息，甚至 Windows 应用程序可以向 Windows 操作系统发送消息，这在前面也提到过。

Windows 系统或者是应用程序有两种传输消息的方法：发送消息（SendMessage）和寄

送消息（PostMessage）。

- 发送消息：发送一个消息时，通信是及时的。直接调用窗口的窗口进程，直到窗口进程为调用函数返回一个结果后，应用程序才结束。
- 寄送消息：寄送一个消息时，把消息发送到窗口应用程序的消息队列中，应用程序一旦空闲，就会搜索消息队列，并在消息队列中处理该消息，即从队列中删除它们，并将它们发送到指定窗口，通信也可能延迟，直到目标应用程序获得处理消息的时间，调用函数发送消息后即返回，但结果只是表示消息寄送是否成功，而不是被调用窗口进程的结果。

消息的发送和寄送过程如图 2-61 所示。

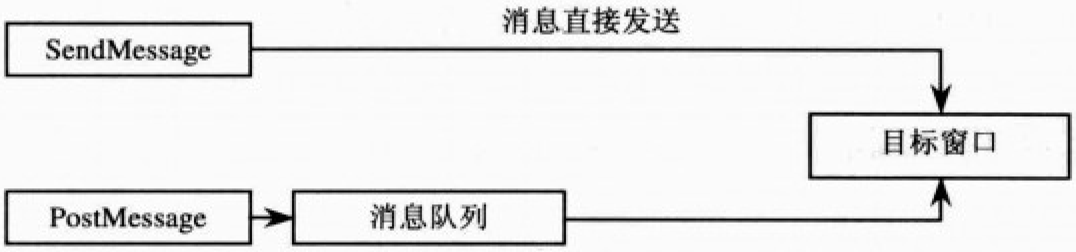


图 2-61　发送消息和寄送消息

而在这个过程中主要的区别就是消息被接收对象收到后是否立即被处理，Windows 为这两种方式提供了多个函数用于消息的发送，常用的函数是 SendMessage()、PostMessage()和 SendDlgItemMessage()。

1. 函数 SendMessage()

函数 SendMessage()用于向目标窗口传送消息，然后一直等到目标线程处理该消息才返回。下面是该函数声明。

```
LRESULT SendMessage (HWND hWnd,     //接收消息的模板窗口
                    UINT Msg,       //消息标志
                    WPARAM wParam,  //第一个消息参数
                    LPARAM lParam   //第二个消息参数
);
```

2. 函数 PostMessage()

函数 PostMessage()用于向目标窗口寄送消息，该函数把消息放在创建目标窗口线程的消息队列中，然后不等消息处理就返回。目标窗口通过调用函数 GetMessage()或者 PeekMessage()从消息队列中取出消息并进行处理。下面是该函数声明。

```
BOOL PostMessage(HWND hWnd,         //目标窗口句柄
                UINT Msg,           //消息标志
                WPARAM wParam,      //第一个消息参数
                LPARAM lParam       //第二个消息参数
);
```

CWnd::PostMessage 提供了对 PostMessage 的简单包装，它首先断言窗口句柄表示一个窗口，然后以窗口句柄为第一个参数调用全局的 PostMessage。这个函数的返回值是一个 BOLL 类型，用来表示消息是否到达了消息寄送队列。

3. 函数 SendDlgItemMessage()

函数 SendDlgItemMessage()用于向对话框中指定的控件发送消息，直到目标控件处理完收到的消息，该函数才返回。下面是该函数声明。

```
LONG SendDlgItemMessage(HWND hDlg,          //目标控件的对话框句柄
                        Int nIDDlgItem,     //目标控件的ID
                        UINT Msg,           //消息标志
                           WPARAM wParam,   //第一个消息参数
                           LPARAM lParam    //第二个消息参数
);
```

MFC 把这 3 个 API 函数同目标窗口句柄封装成 CWnd 窗口类的成员函数，可以向调用它们的窗口对象发送消息，例如函数调用语句 pMyView->SendMessage()表示向指针 pMyView 所指的对象发送消息。

这 3 个 API 封装在 CWnd 类中的成员函数声明为如下。

```
LRESULT SendMessage( UINT message, WPARAM wParam = 0, LPARAM lParam = 0 );
BOOL PostMessage( UINT message, WPARAM wParam = 0, LPARAM lParam =0 );
LRESULT SendDlgItemMessage( int nID, UINT message, WPARAM wParam = 0, LPARAM lParam
= 0 );
```

鼠标和键盘消息通常都是寄送的，而其他消息通常都是发送的。在消息队列中，寄送的消息接受特殊的鼠标和键盘处理。通常，消息应该尽快发送到目标窗口，所以一般都是直接发送消息。

注意：

> 发送消息的返回结果一直要等到目的窗口处理完消息以后才会返回，而寄送消息则是消息到达消息寄送队列后马上返回。

上节介绍了系统定义的消息和应用程序自定义的消息，并且此节上面介绍了对消息的响应和处理。通常我们在 MFC 编程时，都是直接调用 Windows 系统已经定义好的很多标准的消息和消息处理函数，但是从更高层次的编程角度来看，只调用系统本身标准的消息和消息处理函数是不够的。因为程序设计结构的限制或者不同窗口之间数据的同步，开发人员往往需要手工自己定义一些消息。这通常要按照 Windows 格式的要求来自定义消息和消息处理函数。

操作系统保留一定范围的窗口消息标记作为自定义消息。自定义窗口消息的消息标志多大于 WM_USER，这在前面已经提到过。

一旦定义了消息标志，就可以用目标窗口对象的 SendMessage()或者 PostMessage()发送（寄送）消息，以请求目标窗口对该消息进行处理。通常利用#define 语句定义用户自定义消息，例如：

```
#define WM_USER1 WM_USER+0
#define WM_USER2 WM_USER+1
#define WM_MYMESSAGE WM_USER+2
```

也可以调用窗口消息注册函数 RegisterWindowsMessage()来定义一个消息，其好处是不

用考虑消息是否超出范围，并且定义的消息在整个 Windows 系统中都有效，消息可向其他应用程序发送。该函数原型是：

```
UINT RegisterWindowMessage(LPCTSTR lpString);
```

其中，参数 lpString 是要定义的消息名称，调用成功后将返回该消息的 ID 值。

对于自定义消息，需要用户自己手工添加自定义消息映射宏，其格式如下。

```
ON_MESSAGE(message,MemberFun)
```

其中，messge 是消息名，MemberFun 是对应的消息处理函数。例如，假设用户自定义消息 VM_MYMESSGE 的消息处理函数为 OnMyMessage()，则该消息映射宏为：

```
ON_MESSAGE(WM_MYMESSAGE,OnMyMessage)
```

消息映射宏要添加在 BEGIN_MESSAGE_MAP()和 END_MESSAGE_MAP 之间。声明自定义消息处理函数时，与其他消息处理函数一样，必须在函数类型前面加上 afx_msg 标识，如下面代码所示。

```
Afx_msg LRESULT MemberFun(WPARAM wParam,LPARAM lParam);
```

其中参数 wParam、lParam 用于传递消息的两个附加信息，我们对使用自定义窗口消息的用法总结如下。

- 自定义消息的消息标志符必须大于 WM_USER。
- 使用自定义消息请求窗口进行某种特殊的操作。
- 可以利用消息参数携带数据，注意消息参数的内存管理，保证既不发生内存泄漏，又使消息参数有效。

2.4.3 常用消息

由前面介绍的内容可知消息分为 3 类：窗口消息、命令消息、控件通知消息。下面简单介绍一下常见的窗口消息及鼠标消息、键盘消息，如表 2-4 所示是开发人员经常处理的窗口消息。

表 2-4 常见的窗口消息

消息	说明
WM_CLOSE	窗口关闭消息
WM_CHAR	当出现 WM_KEYDOWN 消息时，TranslateMessage()将按键消息翻译后再将 WM_CHAR 消息发送给窗口，这个消息中得到的是可打印字符
WM_CREATE	窗口类的窗口已经创建完毕的消息，此时可以初始化窗口上的一些控件，比如工具栏或状态栏
WM_DESTROY	窗口销毁的消息。此时可清除窗口生命期内的一些内容、释放占用的一些资源
WM_INITDIALOG	此消息针对对话框窗口，此时对话框及对话框上的控件已经全部创建完毕，但窗口还未显示出来，可以此消息内部进行数据的初始化等工作
WM_KEYDOWN	键盘上的按键被按下的消息
WM_KEYUP	键盘上的按键被释放的消息

（续表）

消息	说明
WM_LBUTTONBLACK	当鼠标在窗口客户区时，用户双击鼠标左键的消息
WM_LBUTTONDOWN	当鼠标在窗口客户区时，用户单击鼠标左键的消息
WM_LBUTTONUP	当鼠标在窗口客户区时，用户释放鼠标左键的消息
WM_MOUSEMOVE	鼠标在窗口客户区移动的消息
WM_MOUSEWHEEL	在窗口客户区内滚动鼠标滚轮的消息
WM_MOVE	窗口被移动的消息
WM_PAINT	窗口客户区重绘消息
WM_RBUTTONBLACK	当鼠标在窗口客户区时，用户双击鼠标右键的消息
WM_RBUTTONDOWN	当鼠标在窗口客户区时，用户单击鼠标右键的消息
WM_RBUTTONUP	当鼠标在窗口客户区时，用户释放鼠标右键的消息
WM_SIZE	窗口大小的消息
WM_SHOWWINDOW	隐藏或显示窗口之前，发送此消息。此时窗口已经创建完毕

应用程序能引发许多鼠标消息，常见的鼠标消息如表 2-5 所示。

表 2-5　常用鼠标消息

消息	说明
WM_LBUTTONDOWN	鼠标位于窗口客户区，同时按下鼠标左键
WM_LBUTTONUP	鼠标左键被释放
WM_LBUTTONDBLCLK	鼠标位于窗口客户区，同时双击鼠标左键
WM_MBUTTONDOWN	鼠标位于窗口客户区，同时按下鼠标中间按钮
WM_MBUTTONUP	鼠标中按钮被释放
WM_MBUTTONDBLCLK	鼠标位于窗口客户区，同时双击鼠标中间按钮
WM_RBUTTONDOWM	鼠标位于窗口客户区，同时按下鼠标右键
WM_RBUTTONUP	鼠标右键被释放
WM_RBUTTONDBLCLK	鼠标位于窗口客户区，同时双击鼠标右键
WM_MOUSEMOVE	鼠标在窗口客户区上移动
WM_MOUSEWHEEL	在获得焦点的情况下滚动鼠标滚轮

系统为键盘输入生成的消息十分简单，就只有 3 条消息，如表 2-6 所示。

表 2-6　键盘消息

消息	说明
WM_KEYDOWM	当键盘按键被按下时
WM_CHAR	当用户按键可以转化为有效的 ASCII 码时
WM_KEYUP	当键盘按钮被释放时

2.5 基于对话框的编程

对话框（Dialog Box）是 Windows 应用程序中的一种常用的输入、输出界面，是用户与应用程序进行交互的窗口。对话框也是 Windows 应用程序中使用得最多的用户界面对象，大部分用户的输入都是从对话框中获取的。比如，用户在对话框的文本框中输入文本，从列表框或组合框中进行选择，选择单选按钮或勾选复选框，单击按钮进行确认或取消的操作等。

因为对话框是一个窗口，所有的对话框都是从 CDialog 类派生而来的，而 CDialog 类又是 CWnd 的子类，所以它的生存期符合一般窗口的生存规律：创建、处理窗口消息到销毁。但跟一般的窗口相比，对话框的创建、消息处理和销毁过程有它自己特别的地方。

CDialog 支持两种对话框：模式对话框和非模式对话框。模式对话框打开后，将阻塞主线程的执行，直到被关闭为止，也就是说它是一种以排他的方式工作的对话框。非模式对话框更像一个普通的窗口，它打开后还允许创建它的线程继续处理其他窗口的消息，即它是以非排他的方式工作的。

2.5.1 创建和编辑对话框

在 Visual C++ 2005 中创建一个对话框一般分为 4 个步骤。

STEP 01 利用资源管理器创建一个对话框资源，同时确定哪些控件放入对话框并且设置控件的属性。

STEP 02 利用 MFC 应用程序向导根据对话框资源创建一个 CDialog 的派生类。

STEP 03 编写一些创建对话框的代码，来响应单击某个按钮时出现的对话框。

STEP 04 为对话框上的控件添加消息处理和对话框的初始化代码。

创建对话框资源又可分为几个小步骤：插入对话框资源、设置对话框属性、放置控件、设置控件属性等。

1. 插入对话框资源

我们通过一个简单的 MDI 应用程序添加一个对话框来学习对话框的使用。在上一节中我们创建了 MDI 应用程序 MDIProject，此时我们打开此项目，并选中“视图”菜单中的“资源视图”选项，如图 2-62 所示。用鼠标右键单击“Diaolg”文件夹，从弹出的菜单中选中“添加资源”命令，出现“添加资源”对话框，如图 2-63 所示。然后单击“Dialog”左边的“+”号即可看到要插入的对话框类型。

在弹出的对话框中，会看到对话框的“工具栏”，如图 2-64 所示。在对话框的“工具栏”中有很多 Window 的通用控件。当鼠标放在这些按钮上时就会显示控件名，这些按钮可以调整对话框的控件位置和布局，还可以测试对话框的显式效果。随便双击一个按钮，此按钮就会出现在对话框中，然后按“Ctrl+T”组合键即可看到执行的效果。

2. 设置对话框属性

用鼠标右键单击对话框，选择“属性”选项，将弹出对话框的“属性”面板，如图 2-65 所示。把“属性”面板中的“ID”改为“IDD_MYFIRSTDIALOG”，当应用程序创建对话框时，它就会传递一个对话框资源标识符给 MFC，MFC 则传递一个 ID 给 API 函数，根据此函数读取对话框模板，并创建一个对话框及其所需要的所有控件。我们习惯上对话框的 ID 是以“IDD_”开头的，但是不能和其他 ID 有冲突。然后将对话框的标题即“Capiton”设置为“我的第一个对话框”。也在“Font”中将字体设置为宋体五号。如想修改其他属性，可在列表中修改。

图 2-62 “资源视图”窗口

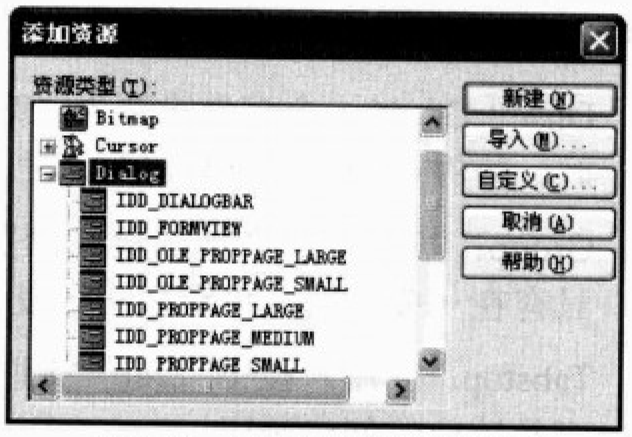


图 2-63 “添加资源”对话框

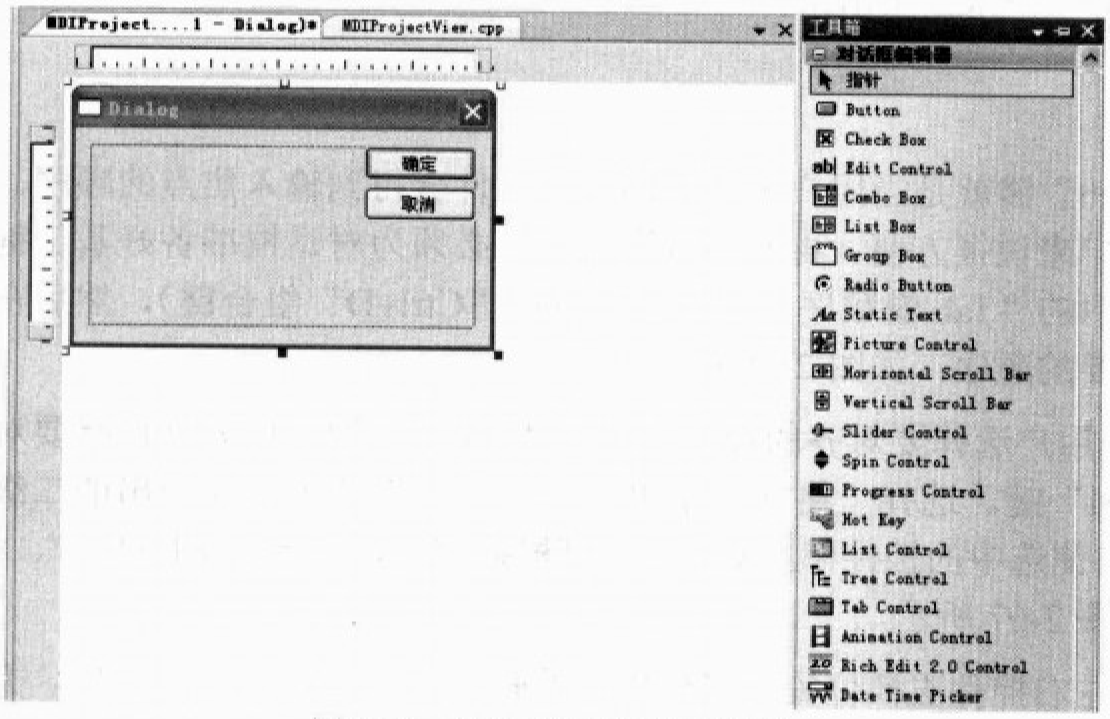


图 2-64 对话框编辑器工具栏

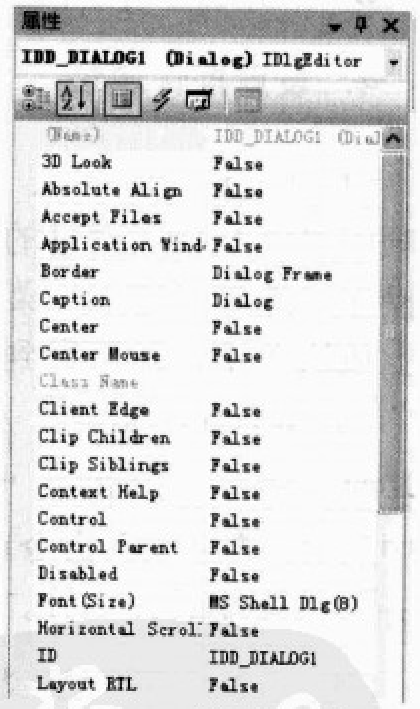


图 2-65 “属性”窗口

当前，对话框上只有系统自带的按钮“确定”和“取消”。如果想添加其他按钮，可以在“工具箱”列表中双击想要加入的按钮，然后可以在“属性”列表中选择需要设置的属性。

3. 添加控件

下面我们就给对话框添加单个常用的控件——按钮、文本框和单选按钮。

- 添加按钮：在“工具箱”列表中单击“Button”按钮，然后在对话框中单击就会出现此按钮（或者双击此按钮，就会出现在对话框的左上角）。可以拖曳边框的蓝色小方块来改变大小。在按钮属性列表中，把标题“Caption”属性改为“显示窗口”。

将按钮的“ID”改为“IDC_SHOWBUTTON”。

- 添加文本框：在“工具箱”列表中单击“Edit Control”按钮，然后在对话框中单击添加控件（也可双击此控件创建），并可以拖曳出控件的合适大小。将文本框的“ID”改成“IDC_EDITINPUT”。
- 添加单选按钮：在“工具箱”列表中单击“Check Box”按钮，然后在对话框中单击添加此控件（也可双击此控件创建），并可以拖曳出合适大小。在单选按钮属性列表中，把标题“Caption”属性改为“请选择”。将“ID”改为“ID_SHOWCHECK”。

在对话框的各个控件属性中，“Disabled”、“Group”、“Tabstop”、“Visible”这 4 个属性对所有的控件都适用。

- Disabled：当一个控件设置为“True”时，即可在对话框中显示；当设置为“False”时，控件被禁用，则显式为灰色。
- Group：控制控件之间的键盘浏览顺序。虽然任意类型的控件都可以用该样式，但只有在一般按钮、单选按钮、复选框上能看到效果。
- Tabstop：当用户按下“Tab”键时，控件就可以得到输入焦点。默认的 Tabstop 的顺序是控件的创建顺序。
- Visible：控件是否可以正常显示出来。

4．设置控件属性

（1）“Tab”键顺序：“Tab”键就是当用户按下此键时，控件会得到输入焦点的顺序，这个顺序对喜欢使用键盘的用户来说很方便。再好的应用程序，必须为对话框准备好基于键盘的操作。选择“格式”菜单中的“Tab 键顺序”命令（或者按“Ctrl+D”组合键），然后在控件上依次单击，即可得到想要的顺序，如图 2-66 所示。

（2）“对齐”方式：首先用户选择要对齐的控件，即按住“Ctrl”键，用鼠标选择要对齐的控件，选中后，松开“Ctrl”键。选择“格式”菜单中的“对齐”选项，在弹出的二级菜单中选择“左对齐”命令可以使选中的控件都左对齐。也可根据需要，选择“居中对齐”、“右对齐”、“顶端对齐”等。如图 2-67 所示。

（3）旋转：如果想要使左右控件更换位置，可使用“旋转”功能。首先选择目标，方法如上。选择“格式”菜单中的“旋转”命令，即可看到旋转后的效果，如图 2-68 所示。用户也可根据需要选择“格式”菜单中的其他命令选项。

最后，可按住“Ctrl+T”组合键来测试对话框设置的效果，如图 2-69 所示。至此，我们完成了对话框的简单创建。

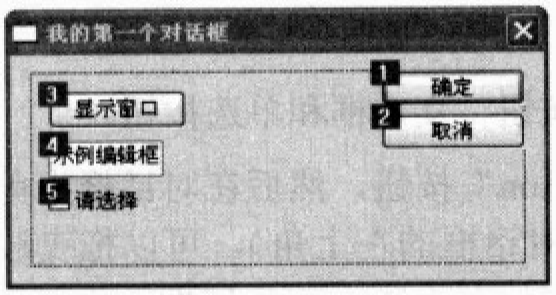


图 2-66　Tab 顺序

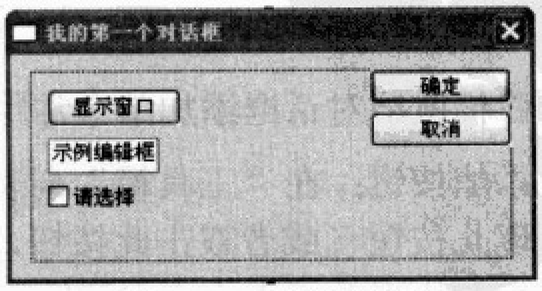


图 2-67　对齐后效果

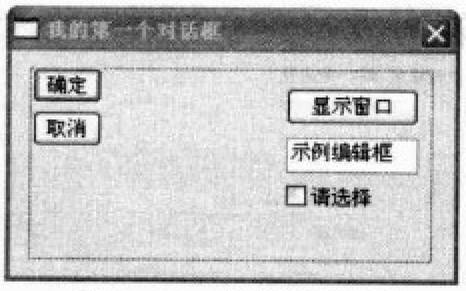


图 2-68　旋转后效果

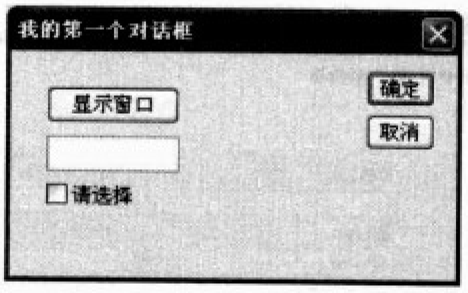


图 2-69　效果图

5. 创建对话框类

新的对话框类都是 CDailog 的派生类，通过这个类，可以初始化对话框并且编写程序来对用户操作进行响应。如果自己动手从零开始编写一个 CDialog 类的派生类，则可能会遇到很多细节上的问题，我们可以直接利用 MFC 类向导很方便地完成这一步。类向导可以帮助用户完成以下任务。

- 从 CDialog 类派生出一个新的对话框类来管理对话框。
- 将 Windows 消息映射到新的对话框类。
- 向新的对话框类中添加成员变量，并且将它们与对话框中的控件联系起来。
- 指定对话框控件和成员变量之间交换数据的方式。

STEP 01 打开对话框编辑器，选择“项目”→“添加类”命令或者双击对话框编辑器，会弹出“MFC 类向导”对话框，如图 2-70 所示。这意味着要用户创建一个对话框类。

STEP 02 在“类名”文本框中输入对话框的类名 CMyFirstDialog。“基类”选项我们选择默认的基类 Dialog，也可以选择新对话框的基类。如果有多个对话框模板，则可以通过“对话框 ID”来选择正确的对话框 ID。在头文件“.h 文件”和源文件“.cpp 文件”的命名中系统自动地生成和类名相同的名称。设置完成后，单击“完成”按钮，进入下一步，如图 2-71 所示。

STEP 03 在弹出的界面中，是对话框的头文件代码，即 MFC 类向导所生成的头文件代码。

```
#pragma once
//CMyFirstDialog 对话框
class CMyFirstDialog : public CDialog
{
DECLARE_DYNAMIC(CMyFirstDialog)                    //声明动态创建宏，此宏支持类的动态创建
public:
CMyFirstDialog(CWnd* pParent = NULL);              //标准构造函数
virtual ~CMyFirstDialog();
//对话框数据
enum { IDD = IDD_MYFIRSTDIALOG };
protected:
virtual void DoDataExchange(CDataExchange* pDX);      //DDX/DDV 支持
DECLARE_MESSAGE_MAP()                              //声明消息映射表，找到相应的消息响应函数
};
```

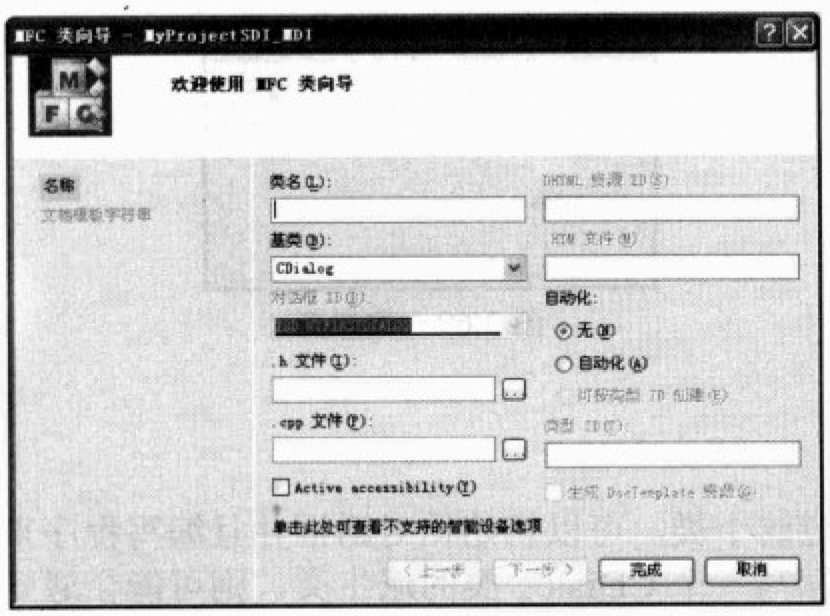


图 2-70 “MFC 类向导”对话框

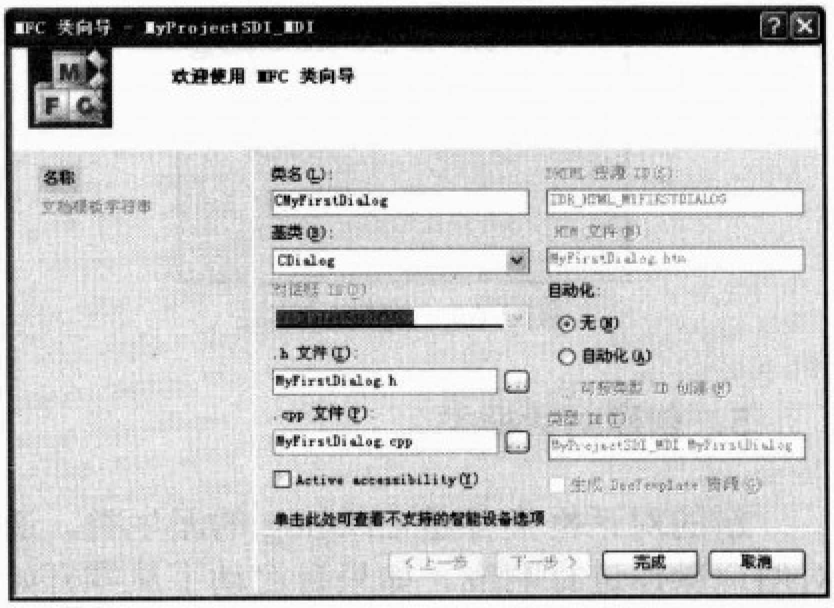


图 2-71 输入类名

在头文件中只定义了一个构造函数和一个虚函数，而现在分别调用了基类的构造函数和虚函数。而在代码中 DoDataExchange()函数则用于完成对话框控件与数据之间的交换。

6. 创建对话框对象

上面我们已经完成了对话框资源和对话框类的创建工作，下一步就是创建对话框对象。

在多数的应用程序中，对话框一般作为菜单命令的响应而出现，所以我们就在处理菜单命令的函数里来创建这个对话框对象。

选择“视图”菜单中的“资源视图”选项，在资源视图面板中，双击“Menu”展开，然后双击 IDR_MAINFRAMES，即出现应用程序框架窗口的主菜单。如图 2-72 所示。

在“帮助”菜单后面的虚线框“请在此处输入”中，单击右键，选中“新插入”选项，输入“对话框（&D）”。这里的“&D”就是用户按住“Alt+D”组合键时，可以打开“对话框”菜单。

在“对话框”下方的“请在此处输入”中输入“模式对话框”，继续输入“非模式对话框”。用鼠标右键单击“对话框”菜单中的“模式对话框”，选择“属性”命令（也可双击“模式对话框”，弹出属性列表），将“属性”列表中的“ID”改为“ID_DISPLAYMODALDIALOG”，这个 ID 不要和其他控件重复。把“Prompt”改为“第一个模式对话框”，这样设置就是当鼠标放到该菜单项上时，状态栏中会显示此提示信息。

用相同的方法设置“非模式对话框”，将属性中的“ID”改为“ID_DISPLAYNONMODALDIALOG”，将“Prompt”改为“第一个非模式对话框”，如图 2-73 所示。

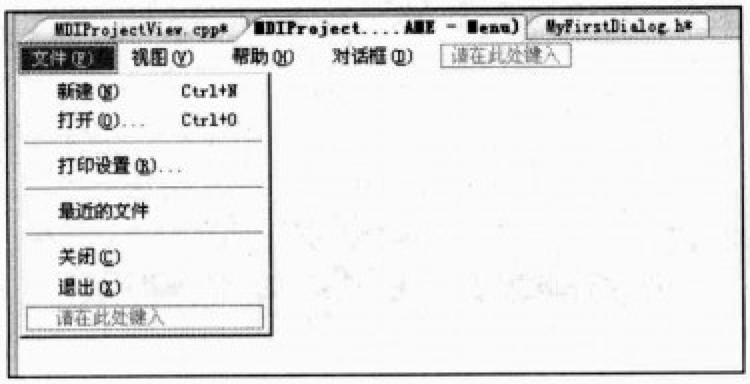


图 2-72 菜单窗口

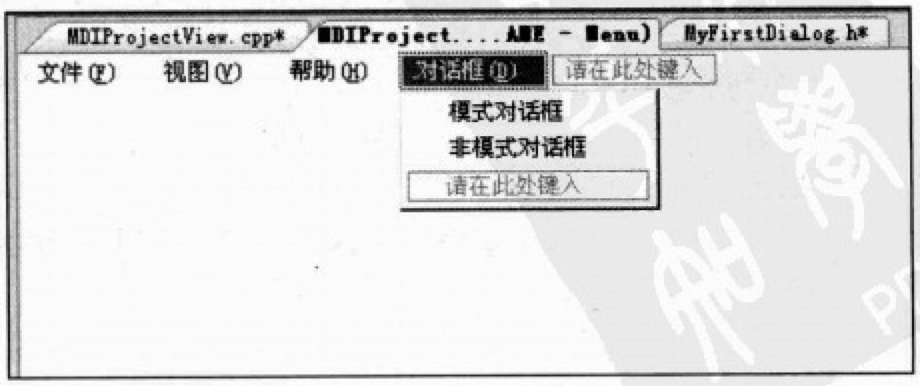


图 2-73 对话框菜单项

然后就要为创建菜单命令处理函数。用鼠标右键单击“模式对话框”，选择“添加事件处理程序”命令，随即会弹出“事件处理程序向导”对话框，在“消息类型”中选择“COMMAND”选项，在“类列表”中选择“CMDIMyProjectView”选项，如图 2-74 所示。

同样，“非模式对话框”也如上设置。设置完成后，单击“添加编辑”按钮，即可进入到消息响应函数代码的编辑窗口。

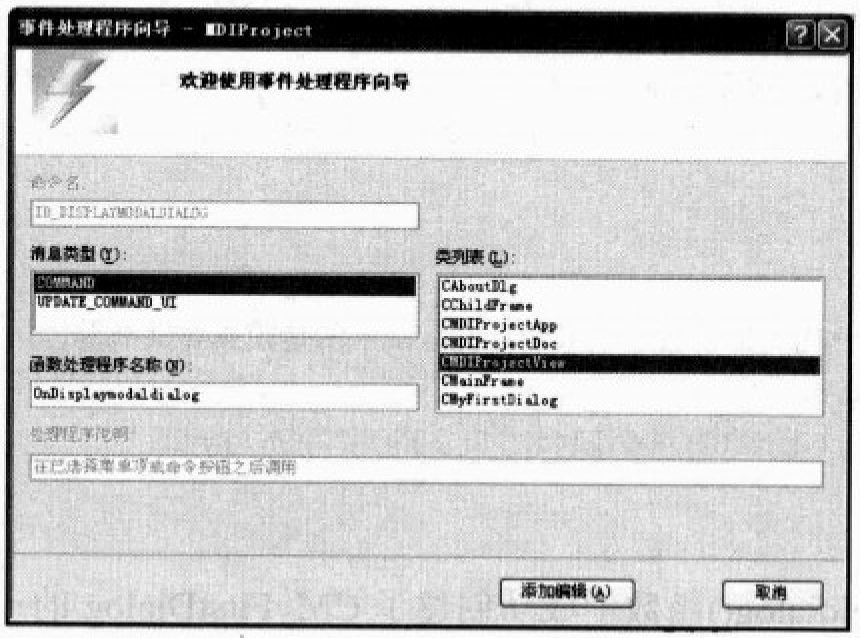


图 2-74　添加事件处理程序

窗口中的代码编辑框架为：

```
void CMDIProjectView::OnDisplaymodaldialog()
{
//TODO: 在此添加命令处理程序代码
}
void CMDIProjectView::OnDisplaynonmodaldialog()
{
//TODO: 在此添加命令处理程序代码
}
```

然后就可以在 OnDisplaymodaldialog()中创建模式对话框了，在 OnDisplaynonmodaldialog()中创建非模式对话框。

注意：

不管创建哪种对话框，都必须要在 CMDIProjectView.cpp 中加入对 CFirstDialog 的引用：

```
#include "MyFirstDialog.h"
```

并且使它位于下列语句的后面。

```
#include "MDIProjectView.h"
```

下面我们创建模式对话框，模式对话框是最常用的对话框，在通常情况下，用户的操作（例如选择一个菜单项或者单击鼠标）都是打开一个对话框，然后在对话框中输入数据，最后关闭对话框。应用程序根据从对话框获得的用户数据，进行下一步操作。

而模式对话框的创建和大多数 MFC 对象一样，都是先实例化一个对象，然后将该对象初始化。在此实例中，我们在 MDIProjectView.cpp 文件中的 CMDIProjectView::OnDisplaymodaldialog()的 TODO 后面添加如下代码。

```
void CMDIProjectView::OnDisplaymodaldialog()
{
//TODO: 在此添加命令处理程序代码
CMyFirstDialog dlg;
int Output=dlg.DoModal();
if(IDOK == Output)
{
MessageBox(L"用户按下去了“确定”键! 演示的是模式对话框编程，谢谢! ",
L"提示",MB_YESNOCANCEL|        MB_ICONINFORMATION|MB_DEFBUTTON1);
}
else
{
    MessageBox(L"用户按下去了“取消”键! 演示的是模式对话框编程，谢谢! ",
L"提示",MB_YESNOCANCEL|
            MB_ICONINFORMATION|MB_DEFBUTTON1);
}
}
```

在 OnDisplaymodaldialog()函数中成功创建了 CMyFirstDialog 的一个对象 dlg，然后通过 CDialog::DoModal()创建了一个模式对话框来显示它。前面提到，模式对话框是以排他的形式工作的，此时用户不能访问其他对象，快捷键也不能使用。模式对话框将输入的消息汇集，并不通过 MFC 正常的消息循环。本实例编译执行结果如图 2-75 所示。

当单击菜单栏中的“对话框（&D）”，选中弹出的“模式对话框”时，即弹出对话框，如图 2-76 所示。当再单击“确定”按钮或者“取消”按钮时，分别弹出的对话框如图 2-77 和图 2-78 所示。

对于模式对话框来说，创建一个对话框对象，首先使用 CDialog 构造函数，并且把资源 ID 作为参数，然后通过调用 DoModal 成员函数来显示模式对话框窗口。一旦 DoModal 函数返回，对话框窗口就被取消。

而非模式对话框的创建就比较复杂了。首先 CDialog 类默认的构造函数（通常没有参数）来创建对话框对象，然后调用 CDialog::Create 成员函数来显示对话框窗口。与 DoModal 函数不同的是，Create 函数把对话框资源 ID 作为参数，并且，在对话框窗口仍然在屏幕上显示的时候，就立即返回。正是这点不同之处，使得非模式对话框编程变得比较复杂，这里就不再赘述了。

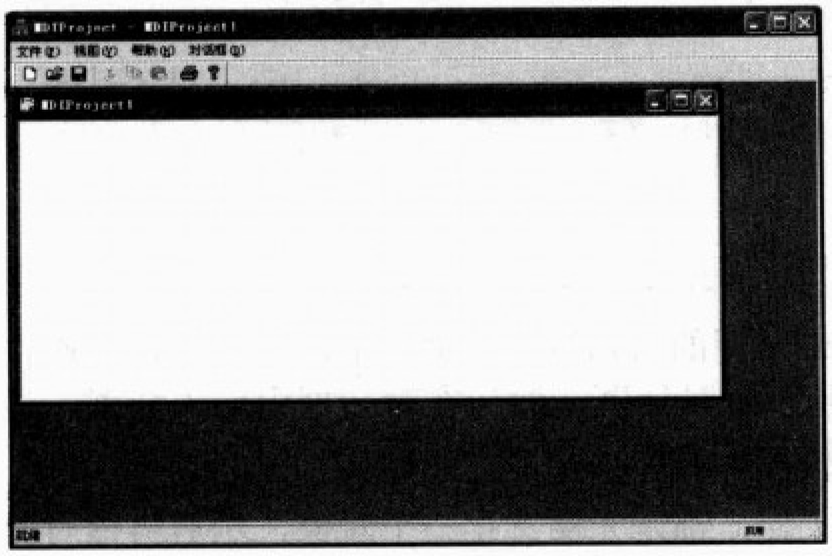


图 2-75　运行结果

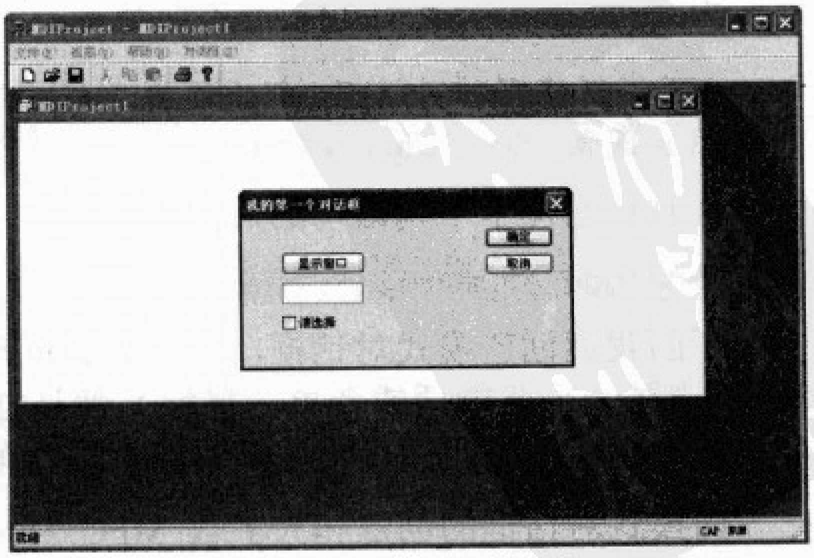


图 2-76　模式对话框

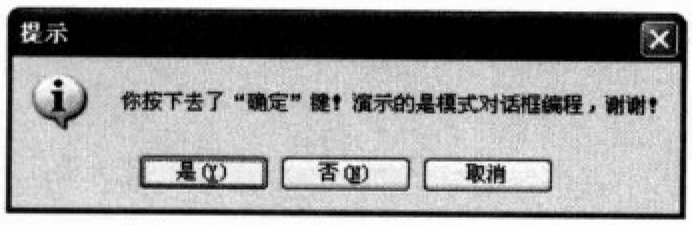


图 2-77 单击"确定"按钮

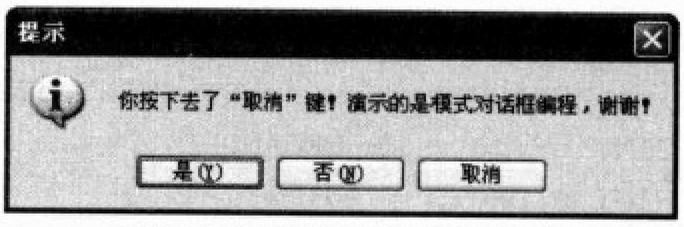


图 2-78 单击"取消"按钮

2.5.2 对话框种类

在编写应用程序的时候，常常利用对话框显示消息。常用的对话框有两种：消息对话框和通用对话框。消息对话框有两个：MessageBox()和 AfxMessageBox()。通用对话框有文件对话框（CFileDialog）、颜色对话框（CColorDialog）、字体对话框（CFontDialog）、打印对话框（CPrintDialog）、查找与替换对话框（CFindReplaceDialog）等。

1. 消息对话框

常用的消息对话框有两个。

1）MessageBox()

类 CWnd 的成员函数 MessageBox()用来创建、显式和操作一个消息框。通常消息框包含了程序定义的消息内容、消息框标题和预定义的按钮、图标。

```
CWnd::MessageBox
int MessageBox( LPCTSTR lpszText,       //消息框中的文本
LPCTSTR lpszCaption = NULL,             //消息框的标题
UINT nType                              //消息框的样式
);
```

如果没有足够的内存以创建消息框，则返回 0。

各个参数的意义如下。

- lpszText 指向一个 CString 对象或以 NULL 结尾的字符串，其中包含要显示的信息。
- lpszCaption 指向一个 CString 对象或以 NULL 结尾的字符串，被用做消息框标题。如果 lpszCaption 为 NULL，则将使用默认的标题"Error"。
- nType 指定了消息框的内容和行为。

说明：

这个函数创建并显示一个窗口，其中包含了应用程序提供的消息和标题，还有预定义的图标和按钮的组合。使用全局函数 AfxMessageBox 代替这个成员函数来为用户的应用程序实现消息框。

2）AfxMessageBox()

AfxMessageBox()是 API 函数 MessageBox()的封装。它有两种定义形式。

```
int AfxMessageBox(LPCTSTR lpszText,
UINT nType = MB_OKCANCEL,
```

```
UINT nIDHelp =0 );
int AFXAPI AfxMessageBox(UINT nIDPrompt,
UINT nType = MB_OKCANCEL,
UINT nIDHelp = (UINT) -1 );
```

如果消息框具有“取消”按钮，那么当按下了“ESC”键或者是单击了“取消”按钮时，就会返回 IDCANCEL 值。如果消息框没有“取消”按钮，那么按下“ESC”键没有任何效果。

函数中各参数意义如下。

- lpszText：指向一个 CString 对象或者以 NULL 结尾的字符串，包含了要在消息框中显示的信息。
- nType：消息框的风格。对对话框应用任何一种消息框风格。
- nIDHelp：消息的帮助上下文 ID。0 表示将使用应用程序默认的帮助上下文。
- nIDPrompt：用于引用字符串表中的字符串的唯一的 ID。

AfxMessageBox()的经常用法是只输入一个参数。由于调试方便，因此它常常用来调试程序时显示调试信息。

表 2-7 中列出了消息框中的图标样式。

表 2-7 消息框中的图标样式

取值	图标样式
MB_ICONHAND，MB_ICONSTOP MB_ICONERROR	
MB_ICONQUESTION	
MB_ICONEXCLAMATION MB_ICONWARNING	
MB_ICONASTERISK MB_ICONINFORMATION	

表 2-8 列举了消息框中常见的按钮样式。

表 2-8 消息框中的按钮样式

取值	按钮样式
MB_OK	确定
MB_OKCANCEL	确定 取消
MB_YESNO	是(Y) 否(N)
MB_YESNOCANCEL	是(Y) 否(N) 取消
MB_RETRYCANCEL	重试(R) 取消
MB_ABORTRETRYIGNORE	终止(A) 重试(R) 忽略(I)
MB_HELP	确定 帮助

根据消息框按下按钮的不同，通常有下表中的可能返回值，如表 2-9 所示。

表 2-9 返回值的意义

返回值	意义
IDABORT	单击了“终止”按钮
IDCANCEL	单击了“取消”按钮
IDCONTINUE	单击了“继续”按钮
IDIGNORE	单击了“忽略”按钮
IDNO	单击了“否”按钮
IDOK	单击了“确定”按钮
IDRETRY	单击了“重试”按钮
IDYES	单击了“是”按钮

2．通用对话框

通用对话框（Common Dialog）是一种 Windows 系统定义的对话框，这种对话框将不同用户的同一类操作过程标准化，以一种通用的形式实现对话框的复杂的功能。对话框的使用非常方便，开发人员不必像对一般对话框那样创建对话框资源和对话框类，就可以在程序中直接使用通用对话框，大大简化了编程工作，并且，开发人员还可以定制通用对话框的外观和性能，使其能够适合于大多数的应用。

为了在 MFC 应用程序中使用通用对话框，MFC 对这些通用对话框以类的形式进行了封装。所有通用对话框类都是从 CCommonDialog 类派生而来的，而 CCommonDialog 类又是 CDialog 类的派生类。表 2-10 列出了 MFC 中的通用对话框类，有关这些通用对话框类的数据结构、构造函数和成员函数请参考 MSDN 帮助。

表 2-10 通用对话框类

MFC 通用对话框类	说明
CColorDialog	颜色对话框，用于选择不同的颜色
CFileDiaolg	文件对话框，用于打开或保存文件
CFontDialog	字体对话框，用于选择不同的字体
CPrintDialog	打印对话框，用于打印和打印设置
CFindReplaceDialog	查找和替换对话框，用于查找和替换文本串
CPagesSetupDialog	页面设置对话框，用于设置页码、页眉、页脚和纸类大小等
ColeDialog	该类及派生来用于生成 OLE 应用程序对话框

使用通用对话框时必须用#include 语句包含 Afxdlgs.h 文件。

在这些通用对话框中，只有“查找和替换”对话框是非模式对话框，其他通用对话框都是模式对话框，它们的使用方式也根据其所属类型的不同而不同。

对于模式通用对话框，通过调用 DoModal()函数打开对话框。对于非模式通用对话框，

通常先用 new 生成对象后，不是调用 CDialog 类的成员函数 DoModal()，而是 CDialog 类的成员函数 Create()装入对话框资源，并创建和显示对话框。下面主要介绍模式通用对话框的使用方法。

在程序中使用一个通用对话框通常遵循以下几个步骤。

STEP 01 声明一个通用对话框类的对象。

STEP 02 修改相应的数据成员，或调用成员函数设置属性值，定制对话框。

STEP 03 调用 DoModal()函数打开对话框。

STEP 04 调用成员函数获取需要的结果。

下面就以颜色对话框为例，来说明利用 CColorDialog 类可以很方便地打开一个颜色对话框，选定一种颜色，并以 COLORREF 为类型返回选中的颜色。COLORREF 是一个双字类型，分别用 8 位数值来保存 R（红）、G（绿）、B（蓝）三基色，这三基色的取值是 0～255 之间的一个整数。COLORREF 的用法如下。

```
COLORREF color(RGB(200,200,200));      //用RGB()函数得到一个COLORREF类型的对象
BYTE red=GetRValue(color);             //分别得到颜色的红、绿、蓝是那个分量
BYTE green=GetGValue(color);
BYTE blue=GetBValue(color);
```

而颜色对话框构造函数的定义如下，接下来给出了颜色选择对话框的使用例子。CColorDialog 的构造函数如下。

```
CColorDialog(COLORREF clrInit=0,
         DWORD dwFlags=0,
         CWnd* pParentWnd=NULL
);
COLORREF crInit(RGB(200,200,200));
COLORREF crCustom(RGB(100,200,200));
CColorDialog dlg(crInit);              //声明颜色对话框类的对象
dlg.m_cc.lStructSize=sizeof(CHOOSECOLOR);
dlg.m_cc.Flags=dlg.m_cc.Flags|CC_FULLOPEN;
dlg.m_cc.lpCustColors=&crCustom;
if(dlg.DoModal==IDOK)     //打开颜色对话框
{
COLORREF color;
Color=dlg.GetColor();     //获取选中的颜色值
CString str;
str.Format(L"用户所选择的颜色的RGB值为：(%d,%d,%d)",GetRValue(color),
                                        GetGValue(color),
                                        GetBValue(color));
   AfxMessageBox(str);
}
```

CColorDialog 最重要的两个成员函数如下。

- GetColor()：得到用户选中的颜色。
- GetSavedCustomColor()：得到用户在“自定义颜色”面板中选中的颜色。

程序的运行结果如图 2-79 所示。

注意：

如果在创建 CColorDialog 类对象 dlg 时，没有使用 CC_FULLOPEN 参数，那么弹出的颜色对话框就只有左半部分，如图 2-80 所示。

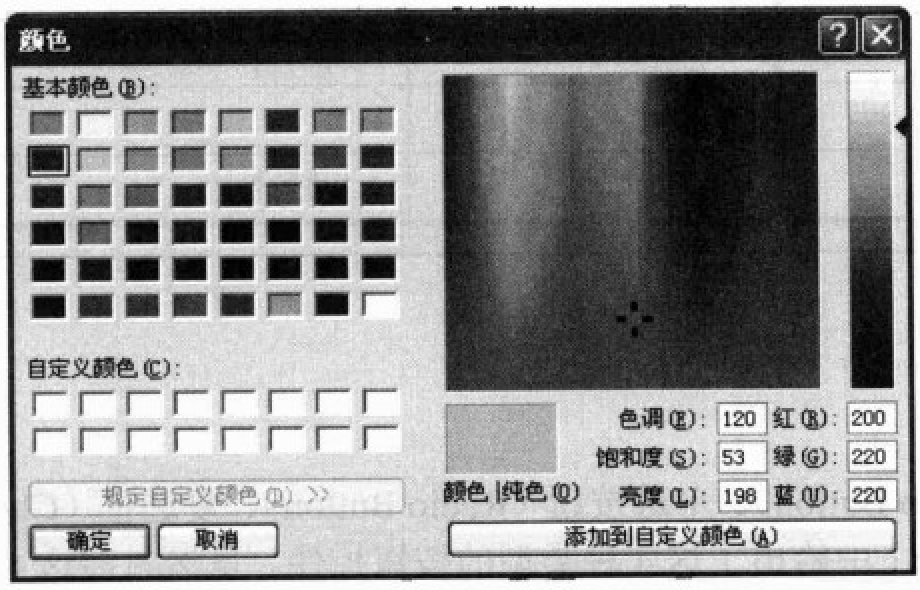


图 2-79 “颜色”对话框

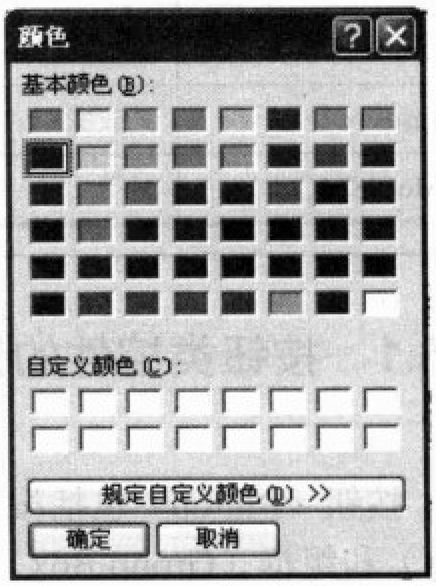


图 2-80 没有全部打开的“颜色”对话框

2.6 常用控件使用技巧

Windows 提供的控件有两类：标准控件和公共控件。标准控件包括静态控件编辑框、按钮、列表框、组合框和滚动条等，标准控件可以满足界面设计的要求。例如，列表框用于选择要输入的信息，复选框按钮用于选择不同的选项，编辑框用于输入用户数据。除了标准控件外，Windows 还提供了一些通用的公共控件，例如滑块、进度条、树形控件、列表视控件和标签控件等，来实现应用程序用户界面风格的多样性。

控件作为程序与用户之间的一个接口，对话框通过控件与用户之间的一个接口，对话框通过控件与用户进行交流。控件实际上也是一个窗口，作为窗口，控件就具有窗口的一般功能和属性，可以调用窗口管理函数来实现控件的移动、显示、隐藏、禁用或可用等操作，也可以重新设置控件的尺寸和风格等属性。

在 MFC 中，MFC 以类的形式对标准控件和公共控件进行了封装，如表 2-11 所示。这些类大部分是从 CWnd 类直接派生而来的。

表 2-11 常用的 MFC 控件类

MFC 类	控件	MFC 类	控件
CAnimateCtrl	动画控件	CProgressCtrl	进度条
CButton	按钮、复选框、单选按钮、组框	CRichEditCtrl	复合编辑框
CComboBox	组合框	CStatic	静态文本、图片控件
CComboBoxEx	扩展组合框	CScrollBar	滚动条

（续表）

MFC 类	控件	MFC 类	控件
CDateTimeCtrl	日期事件选取器	CsliderCtrl	滑块
CEdit	编辑框	CStatusBarCtrl	状态条控件
CImageList	图像列表	CSpinButtonCtrl	旋转按钮
CListCtrl	列表视控件	CTreeCtrl	树形控件
CListBox	列表框	CTabCtrl	标签
CMonthCalCtrl	日历	CToolBarCtrl	工具条控件

2.6.1 按钮类控件的使用

按钮（Button）包括按键按钮（Push Button）、单选按钮（Radio Button）、复选框（Check Box）和组框（Group Box）4 种。图 2-81 中给出了这 4 种类型的按钮控件。虽然封装这 4 种按钮控件的 MFC 类都是 CButton 类，但是它们都具有不同的功能。按键按钮在被按下时会立即执行某个命令，也被称为命令按钮；单选按钮用于在一组互相排斥的选项中选择其中一项；复选框用于在一组选项中选择其中一项或者多项；组框用来作为一组控件的外围边界，可以使一组控件关联在一起，它经常与一组单选按钮或一组复选框一起使用。

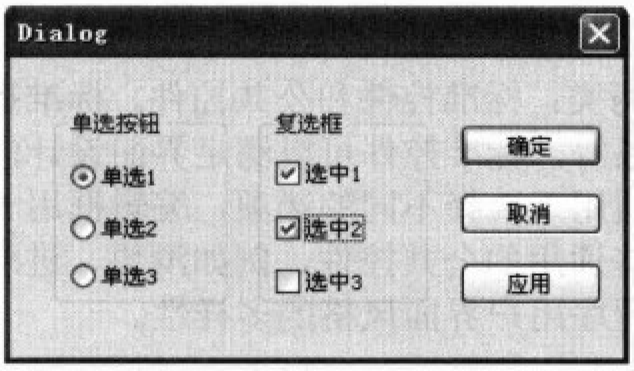


图 2-81　4 种不同的按钮

几乎所有的对话框都使用简单的按键按钮，如经常使用的“确定”按钮。通过控件属性对话框可以设置按钮的风格。

单选框是由一个圆圈和紧随其后的文本标题组成的，当它被选中时，圆圈中就标出一个黑点。通常将一组单选按钮放在一个组框控件中，在一组单选按钮中，第一个（“Tab”键顺序）按钮最重要，其 ID 值可以用于在对话框中为控件建立关联的成员变量。必须为同组的第一个单选按钮设置 Group 属性，而同组的其他单选按钮不能设置 Group 属性。

复选框由一个空心方框和紧随其后的文本组成，当它被选中时，空心方框中就加上一个“√”或“×”标记。不同于单选按钮，在一组复选框中每次可以同时选择多项。处理选中和没选中 2 种状态，复选框还有第 3 种状态，此时复选框显示暗色，它表示该项不可以由用户选择。通过设置 Tri-state 属性得到这种三态复选框。另外，可以通过设置 Push-like 属性使单选按钮具有普通按钮的外观。

按钮控件只能发送通知码为 BN_CLICKED（单击按钮）和 BN_DOUBLECLICKED（双击按钮）的 WM_COMMAND 消息，经常需要编写按钮的 BN_CLICKED 消息处理函数。CButton 类提供了一些成员函数实现对按钮控件对象的控制和管理，如利用成员函数 GetCheck()或 SetCheck()获取或设置单选按钮、复选框的当前状态，利用成员函数 GetButtonStyle()或 SetButtonStyle()获取或改变按钮控件的风格。

2.6.2 文本可编辑类控件的使用

编辑类控件主要包含列表框（List Box）和组合框（Combo Box）。如图 2-82 所示。

为了方便用户在指定的选项中选择需要的选项，这些选项可以采用直接的列表形式显示出来，Windows 提供了能够实现这种功能的列表型控件。列表型控件包括列表框、组合框、列表视控件和树视控件 4 种，其中列表框是一种最简单的列表型控件。列表框（List Box）是一个列出了一些文本项的窗口，常用来显示类型相同的一系列文本信息，如字体和用户名等。与复选框类似，用户可以选择其中一项或多项，但同时列表框中选项的数目和内容可以动态变化，通过编程可以往列表框中添加或删除某些选项。当列表项超出列表框的显示区域时，列表框会自动添加一个滚动条。

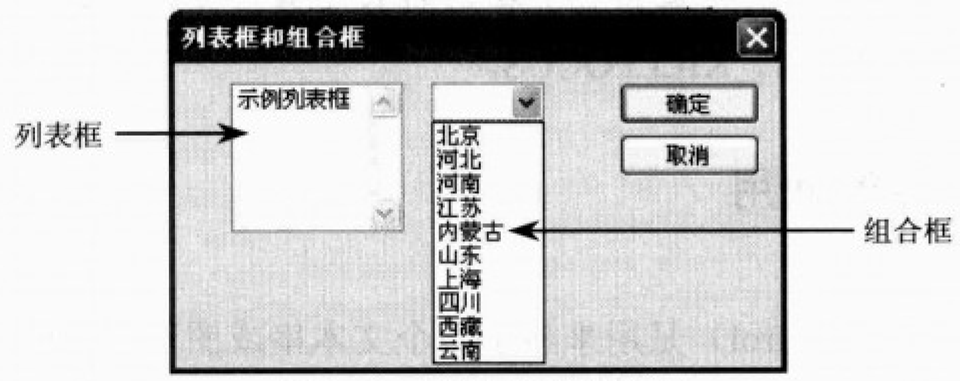


图 2-82　列表框和组合框

列表框有 Single（单选）、Multiple（多选）、Extended（扩展多选）和 None（不选）4 种风格，在控件属性对话框的下拉框中设置以上 4 种风格。默认为单选列表框，表示用户一次只能按下一个选项；多选列表框允许用户在按下“Shift”或“Ctrl”键的同时利用鼠标选择多个选项；扩展多选列表框除了具备多选列表框的功能外，还允许用户在按下“Shift”键的同时利用方向键选择多个选项，而且可以通过鼠标拖曳来选择多个选项；不选风格列表框表示用户不能选择任何选项。

当列表框中发生了某个事件时，如用户双击列表框中的某一项，则列表框就会向其父窗口发送一条通知消息。列表框常用的通知消息有用户双击列表框中的列表项时发送消息 LBN_DULCLK，列表框获得键盘输入焦点时发送消息 LBN_SETFOCUS，列表框失去键盘输入焦点时发送消息 LBN_KILLFOCUS，列表框中的当前选择项发生改变时发送消息 LBN_SELCHANGE。

封装列表框控件的 MFC 类是 CListBox 类，当列表框创建之后，在程序中可以通过调用 CListBox 类的成员函数来实现列表项的添加、删除、修改和获取。

上面介绍了列表框的特点，在使用它时受到一些限制。编辑框运行用户输入文本，但用

户却不能直接选择以前已输入的内容。列表框可列出各种可能的选项，但用户却不能在列表框中输入新的列表项。组合框（Combo Box）吸收了列表框和编辑框的优点，它既可以显示列表项供用户选择，也允许用户输入新的列表项。实质上，组合框是多个控件的组合，包括编辑框、列表框和按钮。

组合框有 Simple（简单组合框）、Drop Dowm（下拉组合框）和 Drop List（下拉列表框）3 种形式，通过控件 Properties 属性对话框 Styles 页面的 Type 下拉列表框设置这 3 种形式。简单的组合框是一个列表框和一个编辑框的组合，列表框总是可见的，被选中的列表项显示在编辑框中。下拉组合框除了含有列表框和编辑框，在编辑框旁还有一个下拉按钮，只有当用户单击下拉按钮时，列表框才会显示出来。下拉列表框除了用户不能在编辑框中进行输入操作外，其他功能与下拉组合框一样。

在组合框的通知消息中，有些是列表框发出的，有些是编辑框发出的。组合框常用的通知消息有关闭组合框的列表框时发送消息 CBN_CLOSEUP，打开列表框时发送消息 CBN_DROPDOWN，双击列表项时发送消息 CBN_DBLCLK，当前选项被取消时（如重新选择其他控件或关闭对话框）发送消息 CBN_SELENDCANCEL，当前选项改变时发送消息 CBN_SELCHANGE，当组合框的编辑框中的文本被修改并且在新的文本显示之前发送消息 CBN_EDITCHANGE，当编辑框中的文本被修改且新的文本显示之前发送消息 CBN_EDITUPDATE，组合框获得键盘输入焦点时发送消息 CBN_SETFOCUS，组合框失去键盘输入焦点时发送消息 CBN_KILLFOCUS。

2.6.3 静态类控件的使用

静态类控件（Static Control）是用来显示一个文本串或图形信息的控件，它包括静态文本控件、图片控件（Picture Control）和组框（Group Box）。静态文本控件用来显示一般不需要变化的文本；图片控件用来显示边框、矩形、图标或位图等图形；组框用来显示一个文本标题和一个矩形边框，通常用来作为一组控件的外围边界，并将一组控件组织在一起。组织和管理静态文本控件和图片控件的 MFC 类是 CStatic 类，而对组框进行管理的 MFC 类是 CButton 类，所以，组框又可以归类于按钮控件。

所有的静态控件默认的 ID 标识都为 IDC_STATIC，如果要为一个静态控件添加成员变量或消息处理函数，则必须重新为它指定一个唯一的 ID 标识。而静态控件是一种单向交互的控件，一般只能在控件上输出某些信息（文本或图形），不用来响应用户的输入，即可以接收消息，基本上不发送消息，只是起到一种标识的作用。

编程时用得最多的是静态文本控件（也叫文本框），它被用来作为其他控件的标题，主要是作为编辑框的标题，因为编辑框本身不能设置标题。

每一个静态文本控件最多可以显示 255 个字符，可以使用“\n”换行符，并可以通过属性列表中的 Styles、Sunken、Modal Frame、Border 和 Client Edge 等不同属性设置控件的不同风格。

有两种方法可以用来创建静态文本控件：一种是使用对话框编辑器添加；另一种是使用

Create()函数的方法创建。这里我们只以对话框编辑器添加控件为例来说明。

例如，在资源视图中，选择“Dialog”中的“IDD_ABOUTBOX”。调整对话框和静态文本控件的大小。在主窗口的最下方，也标识出了静态文本控件所在的位置和大小。选择“工具箱”列表中的“Static Text”，然后在对话框中单击，即可出现静态文本控件。首先在对话框的“Caption”中输入“关于静态文本控件”，立刻看到对话框标题的变化，选中静态文本控件，在属性“Caption”栏输入“\n 版权所有\n 静态文本控件\n”，在“Align text”下拉列表框中选择“Left”项，设置 Sunken 和 Client edge 属性。按“Ctrl+T”快捷键得到测试结果，如图 2-83 所示。

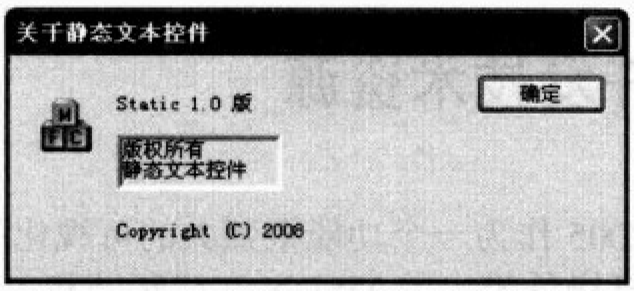


图 2-83　自定义风格的静态文本控件

2.6.4 其他控件

其他的控件主要有滚动条、进度条、滑块、日历等，如图 2-84 所示。这里以滚动条为例进行说明。

当显示的内容超过窗口的大小范围时，需要为视图添加滚动条。同样在一些控件中，当显示的内容超过控件区域范围时，这些控件可以自动增加一个滚动条（Scroll Bar）。实际上，滚动条可以作为一个独立的控件使用，其主要作用是通过可视化的滚动操作实现程序设计所要求的功能，如滚动显示数据内容、在一个有效范围内选取合适的数值等。

滚动条分为垂直滚动条和水平滚动条两种，除了滚动的方向，它们的组成都是一样的。滚动条两端有两个箭头按钮，中间有一个可沿滚动条方向移动的滚动块。当用户移动滚动条时，滚动条就会向父窗口发送水平滚动消息 WM_HSCROLL 或垂直滚动消息 WM_VSCROLL。滚动消息含有通知码，表示用户对滚动条的操作方式，如 SB_LINEUP 和 SB_LINEDOWN 表示向上或向下滚动，SB_THUMBTRACK 表示拖到滚动块。封装滚动条控件的 MFC 类是 CScrollBar 类。滚动条编程的主要步骤如下。

STEP 01 向对话框中添加滚动条控件。

STEP 02 为滚动条映射成员变量。

STEP 03 在对话框的初始化成员函数 OnInitDialog()中，通过滚动条控件成员变量调用 CScrollBar 类的成员函数，设置滚动条的滚动范围和滚动块的位置。

STEP 04 利用向导添加滚动条处理函数，根据传递给函数的消息通知码参数 nSBCode、滚动块位置参数和其他参数进行相应的处理，实现具体的功能。

Windows 公共控件中常用的旋转按钮和滑块控件继承了滚动条的许多功能，并且 MFC 又提供了滚动视图类 CScrollView，这使得滚动条控件一般很少使用。因此，下面就不给出滚

动条编程的例子了。

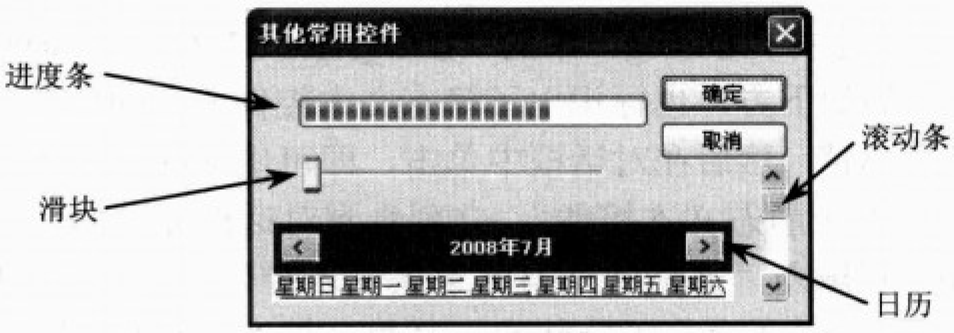


图 2-84　其他常用控件

2.7 Visual 2005开发技术资源

Microsoft Visual Studio 2005 作为一个功能很强大的可视化应用程序开发环境，它具有极其丰富的开发技术资源，包括微软推出的 MSDN 在线帮助和丰富的网络资源等。

2.7.1 MSDN在线帮助

MSDN 的全称是 Microsoft Developer Network。这是微软公司面向软件开发者的一种信息服务。如图 2-85 所示，这是 Visual C++ 2005 提供的访问帮助系统的新工具，它一般显示了开发任务的组织结构，包含丰富的技术编程信息，包括示例代码、文档、技术文章和参考指南。

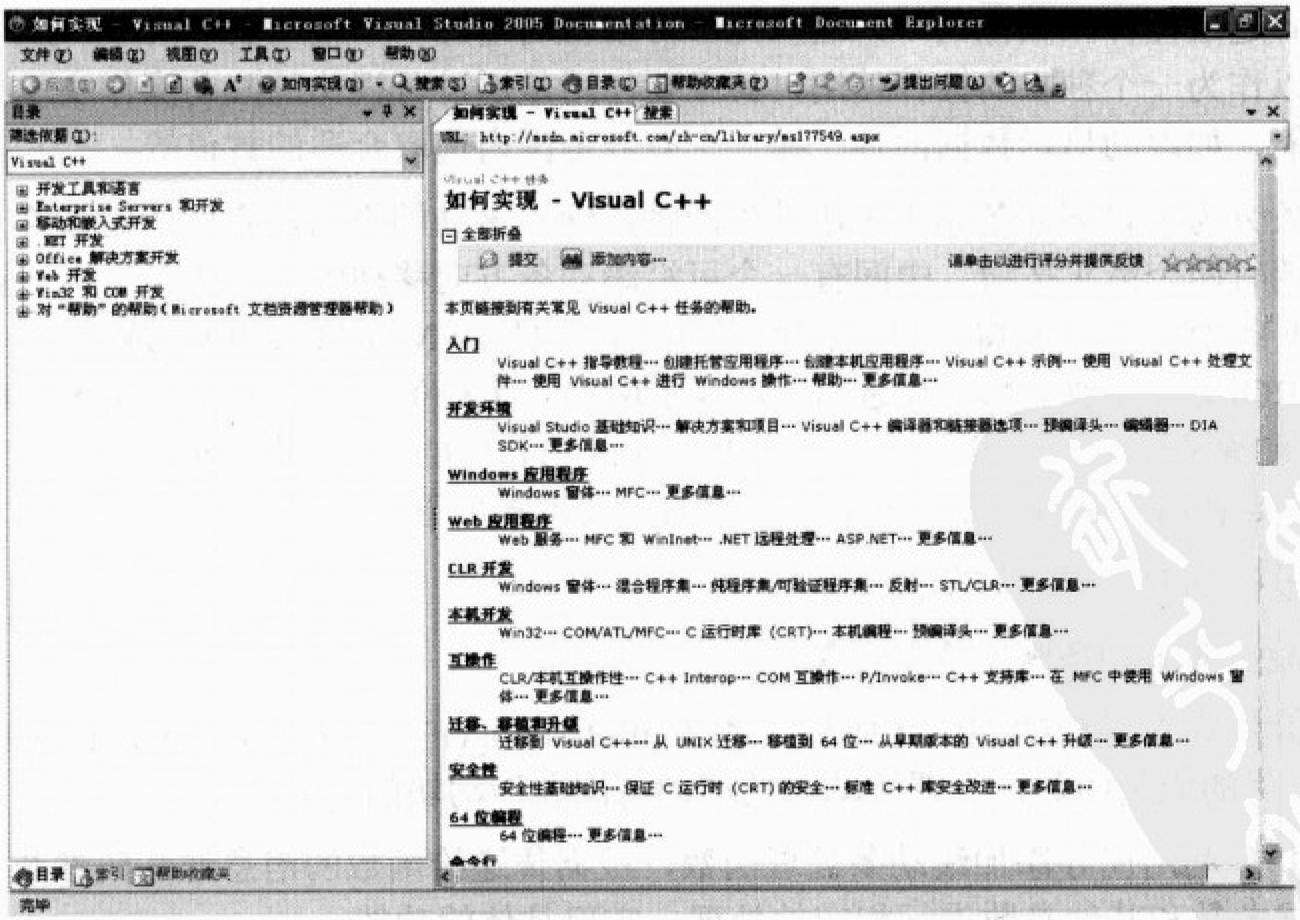


图 2-85　MSDN 帮助

选中“帮助”菜单中的“搜索”选项，可以看到右侧有“MSDN Online”、“Codezone 社

区”、“问题”、“本地帮助”4个选项，可以得到相关链接的页码信息，如图2-86所示。同时也可以选择“工具”菜单中的“选项”选项，得到“帮助”信息，如图2-87、2-88和2-89所示。

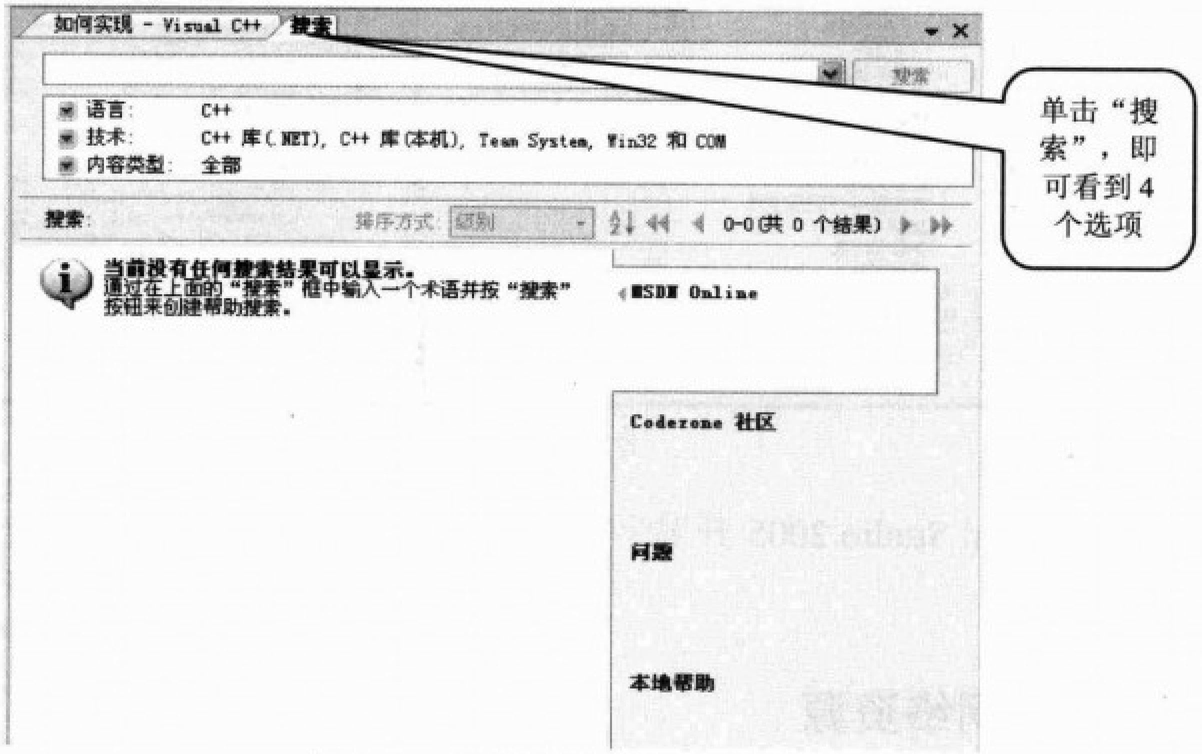


图2-86 搜索帮助信息

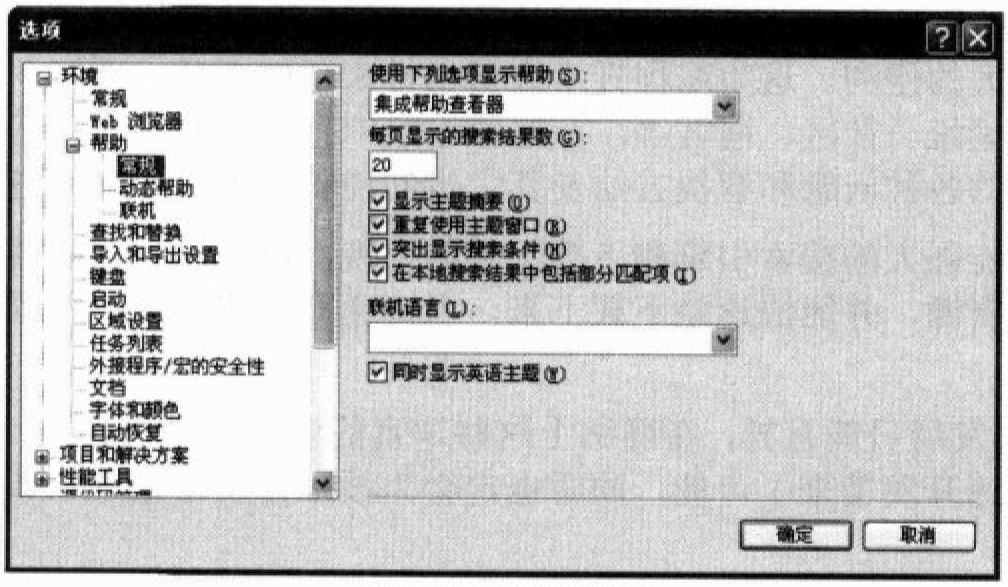


图2-87 常规帮助

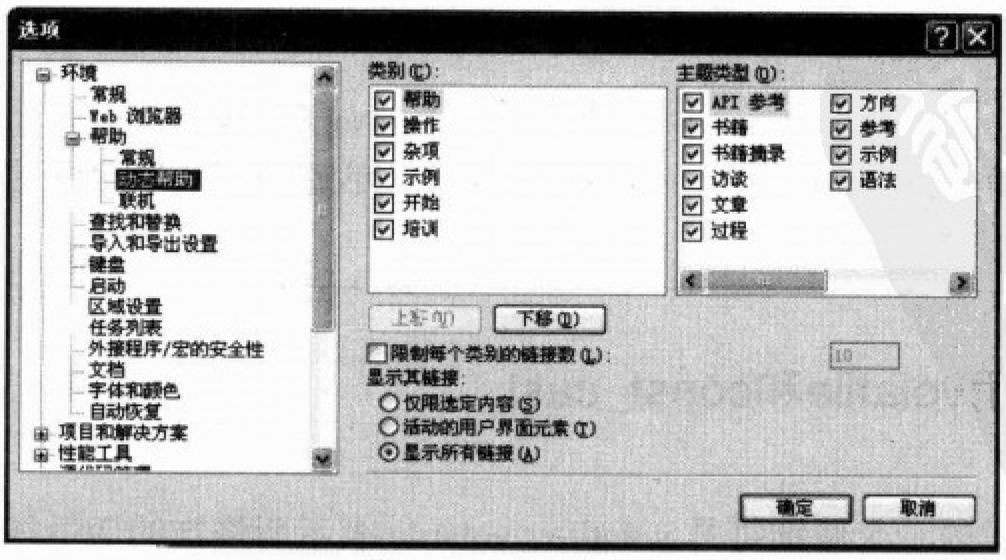


图2-88 动态帮助

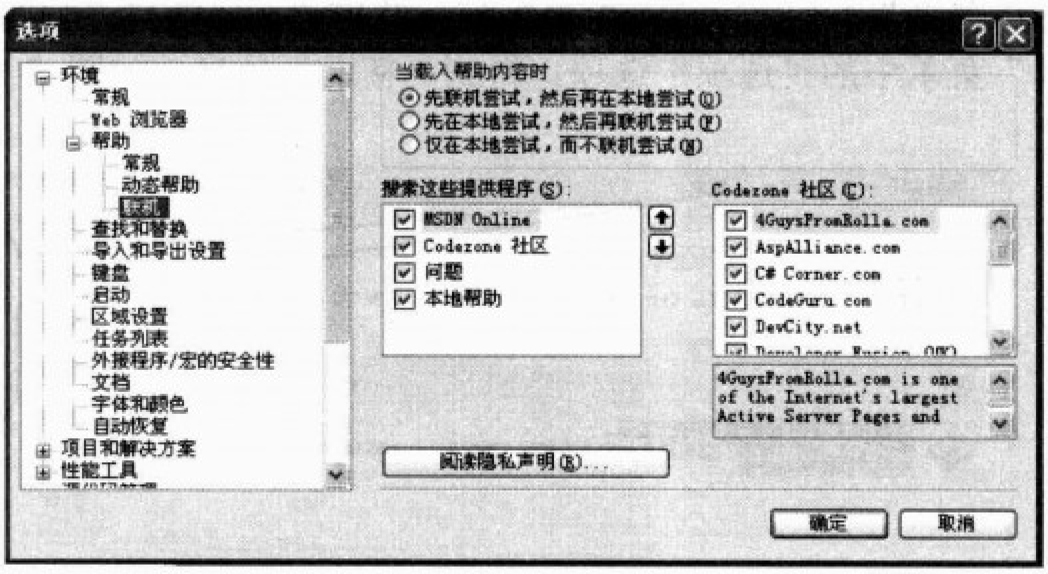


图 2-89　联机帮助

可以看出 Visual Studio 2005 开发平台所提供的一些强大的帮助功能，它们为开发人员带来了很大的方便。

2.7.2　丰富的网络资源

随着 Internet 的迅速发展，伴随着我们的是越来越丰富的网络资源，这些丰富的网络资源为我们带来了极大的便利。这为编程技术人员也带来了好消息，现在，在网络上出现了很多专业的编程技术论坛、社区、博客等，在这些论坛中有丰富的共享技术资料和开发人员在开发过程中所遇到的各种问题和解决方法及其宝贵的经验，可以即时地进行沟通和交流。

如今，随着功能强大的搜索引擎和下载工具的出现，可以通过这些搜索引擎和下载工具快捷地搜到想要的资源，并能快速地下载下来，为快速高效的开发应用程序带来了前所未有的便利。

Internet 技术的发展日新月异，在网络上这些丰富的资源会越来越多，也会越来越全，这为编程技术人员快速开发带来了可能，同时也定会为技术开发带来一个新时期。

2.8　Visual 2005开发的常见问题

在学习 Visual C++ 2005 的过程中经常会碰到各种问题。鉴于此，下面列举了在开发过程中经常遇到的典型问题和解决方法，帮助读者及时、迅速地解决所遇到的问题，并从中得到一些解决问题的技巧。

2.8.1　如何使用volatile和const_cast

与 const 相对应的一个修饰符是 volatile。volatile 表示所修饰的变量能被该进程外的进程访问，所以 volatile 对于声明共享内存或全局数据区域是很有用的。如果同时使用 const 和 volatile 来修饰一个变量声明，则表明该程序不能修改此变量，而其他的进程则可以修改。

Const_cast 操作符可以显示转换指向任何类型的对象的指针及指向数据成员的指针，转换将去掉 const、volatile、_unaligned 属性。表达式：

```
const_cast<type-id>(expressio)
```

Const_cast 属于强制类型转换操作符，但转换不能改变指针指向对象的类型及数据成员的类型。

2.8.2 如何构造unicode版本的程序

只需要在项目设置里面定义一个宏 unicode，VC 就能将项目编译成 unicode 版本。对于 MFC 程序而言，如果要能构造 unicode 版本的程序，需要 MFC42u.lib（Release 版）和 MFC42ud.lib（Debug 版）。

2.8.3 如何使用.def文件

.def 文件常见的用途有定义 DLL 的输出函数和输出数据、定义节的属性及制定栈的大小等。实际上，可以通过.def 文件直接生成.lib 文件。而高版本的 VC 却没有这个功能。有时可能需要在程序中用到 DLL 中的某些非文档化的函数，如果生成一个对应的 LIB 文件，那么就可以在程序中通过链接此 LIB 文件而实现到 DLL 的动态链接。比如，NTDLL.DLL 中有很多函数都是未公开的，可是却希望在程序中使用某些函数，那么可以先用 DEPENDS 工具来查看 NTDLL.DLL 的输出，然后编写一个正确的.def 文件，并使用 LIB 工具来生成对应的 LIB 文件。

2.8.4 如何正确编写Win32 DLL

动态链接库提供了一种代码共享的机制，从而可以有效地减小程序文件的尺寸和减少不必要的重复工作。32 位的 Windows 是一个多任务多线程的操作系统，因此要正确地编写一个 DLL，必须要考虑几个方面的问题。

关于入口点 DllMain 的说明如下。

DLL 的入口点函数通常是 DllMain（也可以通过链接开关指定成别的名字），这个函数的原型如下。

```
BOOL WINAPI DllMaim(
    HINSTANCE hinstDLL          //DLL模块句柄
    DWORD fdwReason             //调用原因
    LPVOID lpvReserved          //保留
```

DllMain 函数不是必须的。如果不需要任何初始化，或者不处理任何事物（比如资源 DLL），就可以不要 DllMain 函数。另外 DllMain 的返回值标志了 DLL 初始化是否成功，如果返回 False，则系统认为初始化失败，DLL 将不被继续加载。

DLL 中可以有两种函数：一种内部函数，通常是一些辅助函数，只能被本模块内的其他

函数调用；一种是外部函数，可以被加载 DLL 的外部模块调用。要从 DLL 中输出一个关键字_declspec（dllexport），则需通过链接开关/EXPORT 指定输出函数。

这里所讨论的变量输出，是针对 DLL 中的全局变量而言的，局部变量不可能输出。输出的全局变量会在每个进程中都有一份复制。要从 DLL 中输出一个全局变量，只需要在变量名称前面使用关键字_declspec（dllexport），而在要引用该变量的进程中使用关键字_declspec（dllimport）即可。

2.8.5 如何编写和使用资源DLL

纯资源 DLL 就是只包含资源的 DLL。由于 DLL 能被共享，因此纯资源 DLL 的优势就在于其资源能够被共享。如果多个程序都要用到相同的资源，那么就可以考虑使用编写纯资源的 DLL 了。另外，由于使用纯资源 DLL 可以实现资源与应用程序分离，所以可以提供不同版本的资源 DLL 来改变应用程序界面，而无须修改程序本身。

纯资源 DLL 的编写是很简单的，读者无须写任何代码，只需要将资源脚本文件和相应的资源文件（位图、图标等）加入到 DLL 项目中，让 VC 将资源编译并链接到 DLL 中即可。

要在应用程序中使用 DLL 中的资源也很简单，首先调用 LoadLibrary 函数来获取资源 DLL 的模块句柄，然后以此为参数调用 FindResource 来寻找各种资源，找到后就可以对资源进行其他操作了。

2.8.6 如何实现对话框的数据交换

要将控件上的数据保存到成员变量上，可以通过 DoDataExchange()来实现。而对话框类的 DoDataExhange()的原型是：

```
Void CMyFirestDialog::DoDataExchange(CDataExchange* pDX)
{
   CDialog::DoDataExchange(pDX);
   DDX_Text(pDX,IDC_EDIT1,m_edit1);
}
```

这样的函数交换是一个相互的过程，当初始化应用程序时，将初始化过的成员变量的数据复制到控件上；当调用 UpdataData()初始化进行更新时，则将控件上的数据保存到成员变量中。

设置 DDX 列表是通过“添加成员变量向导”来实现的，随着编程经验的增加，读者可以直接在 DoDataEchange()内插入 DDX 列表以完成数据交换。常见的数据交换函数有 DDX_Check()、DDX_CBString()、DDX_CBIndex()、DDX_CBStringExact()、DDX_Ridio()、DDX_LBString()、DDX_LBStringExact()、DDX_Scroll()、DDX_Slider()、DDX_MonthCalCtrl()、DDX_DateTimeCtrl()等。

这些函数都是以“DDX_”开头的，后面跟的就是通用控件类型，比如 DDX_CBString() 表示与组合框的数据交换。数据交换函数的参数也是一致的，第一个参数总是 CDataExchange*类型的指针，第二个参数是控件的 ID，第三个参数是成员变量。

最后，还可以不用成员变量来访问控件，是通过调用 CWnd::GetDlgItem()来实现的。DDX还适用于复选框、单选按钮、列表框和组合框等控件。

2.9 本章小结

本章介绍了如何在 Visual 2005 开发平台中利用向导来生成传统的控制台应用程序、生成基于 MFC 的应用程序，其中在生成基于 MFC 的应用程序中详细地介绍了静态链接库和动态链接库的知识及其应用程序的生成过程。然后利用 Visual C++ 2005 中的向导完成了应用程序的基本开发。MFC 是在 Windows API 的基础上做了面向对象的封装，MFC 在应用程序开发中起到了举足轻重的作用，随后介绍了 Windows 中的消息和事件，并给出了 MFC 中的消息机制。对话框是 Windows 应用程序中使用得最多的界面对象，随后介绍了对话框的编程，以及一些常用控件的使用方法。Visual C++ 2005 开发平台也为开发人员提供了丰富的技术资源，比如 MSDN 在线帮助、网络资源等。在此开发平台上进行开发往往会遇到诸多问题，在本章最后列举了在用 Visual 2005 开发平台时所遇到的常见问题及解决方法。

第 2 篇

音频开发

本篇讲解 DirectSound 音频开发基础，并通过 4～6 章的实例对 DirectSound 开发技术进行讲解。由于 Direct Sound 在音频开发中具有优势，因此此部分首先对 DirectSound 的基本结构、具体使用方法、开发环境配置及音频应用程序的调试技术进行了详解，接着通过具体的案例详细介绍了基于 Visual C++ 2005 的音频开发技巧和方法，每个实例都按照功能分析、设计思路、编程原理和实例详解 4 个步骤进行阐述。本部分是音频开发的重要内容，通过此部分的介绍，读者可以理解和掌握 DirectSound 开发的步骤和方法，同时读者也可以试着设计自己的音频应用程序。

- 第 3 章　DirectSound 开发基础
- 第 4 章　打造自己的音频播放器
- 第 5 章　语音 3D 特效器制作
- 第 6 章　对拾音设备录制自己的语音文件

第 3 章

DirectSound 开发基础

DirectSound 是个功能强大的应用架构，同时也是个功能强大的音频开发工具。虽然 DirectSound 是一个比较底层的部件，但是它提供了丰富的接口函数，可实现 WAV 声音数据的播放控制及多个声音的混音，同时也可以模拟真实的 3D 音效。所以在音频开发过程中选用 DirectSound 进行开发是个不错的选择。本章将详细介绍 DirectSound 编程的各个步骤和原理，包括缓冲区的创建、声音的播放、音效的使用及 3D 声音的使用。

本章重点

- Microsoft DirectSound SDK
- DirectSound 的详细使用方法
- 开发环境配置
- 调试音频处理程序
- DirectSound 开发的常见问题

PDG

3.1 Microsoft DirectSound SDK

DirectSound 是 Microsoft 多媒体技术 DirectX API 的音频(WaveAudio)组件之一。DirectX 是微软开发的一组多媒体编程接口，它以 COM 组件的 DLL 形式提供 Windows 平台的 API 函数调用，解决了高层软件与硬件的隔离问题。

3.1.1 比较DirectSound与DirectMusic

Microsoft DirectX Audio 为播放集成的音乐和声音效果提供了新的体系结构。尽管仍然使用名称 Microsoft DirectSound 和 Microsoft DirectMusic，但在它们之间已没有明显的区别。

DirectX Audio 有两个组成部分：DirectSound 和 DirectMusic，在这两个部分中 DirectMusic 的变化比较大（与前面的版本相比较），DirectSound 几乎跟以前的一样。

DirectSound 是处理数字声音的最主要的部件。DirectMusic 处理多种文件格式，比如 MIDI、本身自带格式、WAV 文件等，最后把音符信号传给 DirectSound 来处理，如图 3-1 所示。

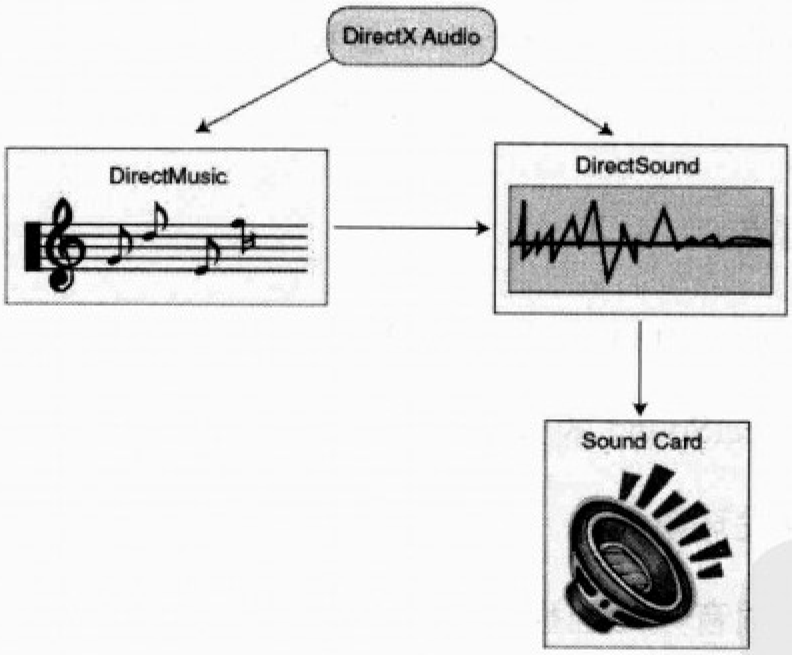


图 3-1　DirectMusic 和 DirectSound 关系

DirectMusic 为大多数交互性应用程序提供完全的声音效果支持的应用程序接口（API）。DirectMusic 既可以播放音乐，也可以播放非音乐。同时，DirectSound 也是一个应用程序接口（API），它可以对声音的硬件资源进行底层控制，也可以让应用程序用很少的系统资源来延时播放声音，这种功能用于在某些游戏中的一些特定情节中实时播放一些声音。

DirectSound 应用程序接口可以单独使用，它可以播放 WAV 文件，也可以和 DirectMusic 一起使用，可以用 DirectSound 操作由 DirectMusic 管理的声音缓冲区。如表 3-1 所示是对这 2 个 API 所提供的功能的比较。

表 3-1 DirectMusic 和 DirectSound 功能对比

功能	DirectMusic	DirectSound
播放 WAV 声音	可以	可以
播放 MIDI 声音	可以	不可以
播放 DirectMusic Producer 声音片段	可以	不可以
加载文件内容并且管理对象	可以	不可以，支持简单代码
在运行时控制声音参数	可以	不可以
管理提示音的时标	可以	不可以
使用可下载的声音（DLS）	可以	不可以
设置声音的音量、音调面板	可以，通过 DirectSound API	可以
设置多声道音量	可以	不可以
实现 3D 声音	可以	可以
音效（DMOs）	可以，通过 DirectMusic Producer 内容和 DirectSound API	可以
为 mix-in 音效进行链式缓冲区处理	可以，通过 DirectMusic Producer 内容	不可以
捕获 WAV 声音	不可以	可以
实现全双工	不可以	可以
捕获 MIDI 声音	可以	不可以
控制硬件缓冲区的分配	不可以	可以

3.1.2 DirectSound SDK能做什么

DirectSound 封装了大量的音频处理 API 函数，它可以提供快速的混音、硬件加速功能，并且可以直接访问相关设备，当然，最主要的是它提供的功能与现有的设备驱动程序保持兼容性。

DirectSound 的优势当然和 DirectX 的其他组件一样——速度，它允许用户最大效率地使用硬件，并拥有良好的兼容性。DirectSound 允许对 WAV 格式的声音文件进行捕获、重放，也可以通过控制硬件和相应的驱动来获得更多的服务。总之，它具有音频编程人员所需要的两个特性：速度快和空控制性。DirectSound 和 Windows 多媒体 API 函数相比，有如下优点。

- 很方便地了解硬件能力，并且根据当前计算机硬件配置来决定最好的解决方法。
- 弥补驱动程序的不足——通过属性设置以便硬件能力可以完全发挥，即便是驱动程序没有很好地支持该功能。
- 当声音重现时，延时时间短，小于 20 微秒。
- 实现 3D 定位效果。
- 可以自动地根据复杂的输入数据格式匹配输出格式。
- 可以在多个声源情况下实现混音。

DirectSound 使程序花费很少的延时就可以播放声音文件，而且可以应用程序对声音硬件

资源进行高级控制。用 DirectSound 可以实现下列功能。

- 播放 WAV 格式的声音文件或资源。
- 同时播放多路声音。
- 在硬件控制的缓冲区中指定声音播放的优先级。
- 在 3D 环境中设置某个声音的位置。
- 在声音中增加一些效果，如回声和齐声等，并且可以改变这些效果的参数。
- 从麦克风或者其他拾音设备中捕捉声音并保存为 WAV 文件。

3.1.3 如何获取DirectSound SDK

虽然说 Visual C++是网络和多媒体底层开发的主要工具，并且功能强大。但是单纯以 Visual C++本身是无法设计出精良且高效率的多媒体程序的，目前的音/视频开发，特别是游戏开发技术，都运用了 DirectX 来加强多媒体功能。

DirectX SDK（DirectX Software Development Kit）是微软开发出的一套应用程序编程接口（APIs），目前最新版本已升至 10.0。在 DirectX 中包含了各种组件用来处理 2D 和 3D 声音、视频、网络联机，以及控制各类输入装置，可以说它是多媒体、游戏程序中不可或缺的主角。本章只对 DirectSound 在音频开发方面的应用做详细的阐述。

要用 DirectSound 进行音频开发，不仅要有功能强大的 VC 作为主要开发工具，同时还要和 DirectX 组件中的 DirectSound 联合来开发。那么，如何获取 DirectSound SDK 呢？下面就简要说明如何获取 DirectSound SDK。因为本章中我们将用 DirectX 9.0 SDK 来开发，所以以 DirectX 9.0 SDK 为例。

STEP 01 因为 DirectX 是 Microsoft 推出的主要用于多媒体应用程序和游戏程序的开发，所以可以亲自访问微软的网站 http://www.microsoft.com 来获取更多的 DirectX 的相关信息。微软的主页如图 3-2 所示。

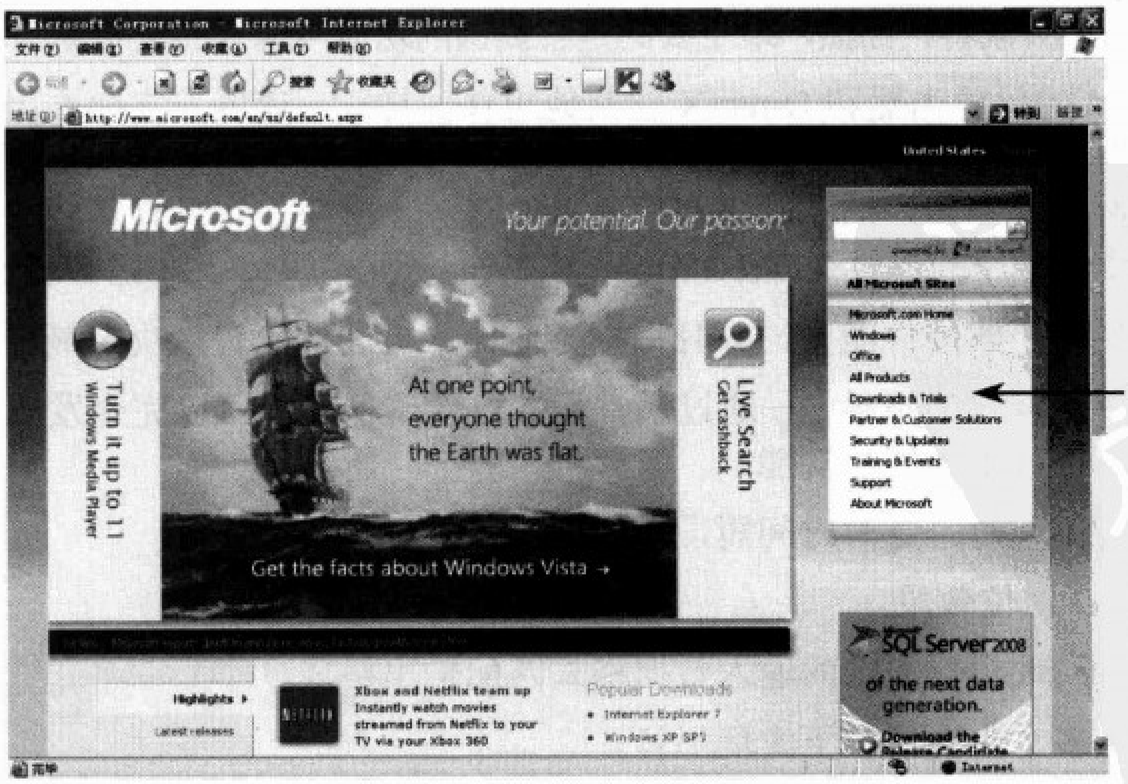


图 3-2 Microsoft 主页

STEP 02 进入 Microsoft 主页后，选择右边列表中的“Downloads”进入 Microsoft 的下载中心，如图 3-3 所示。

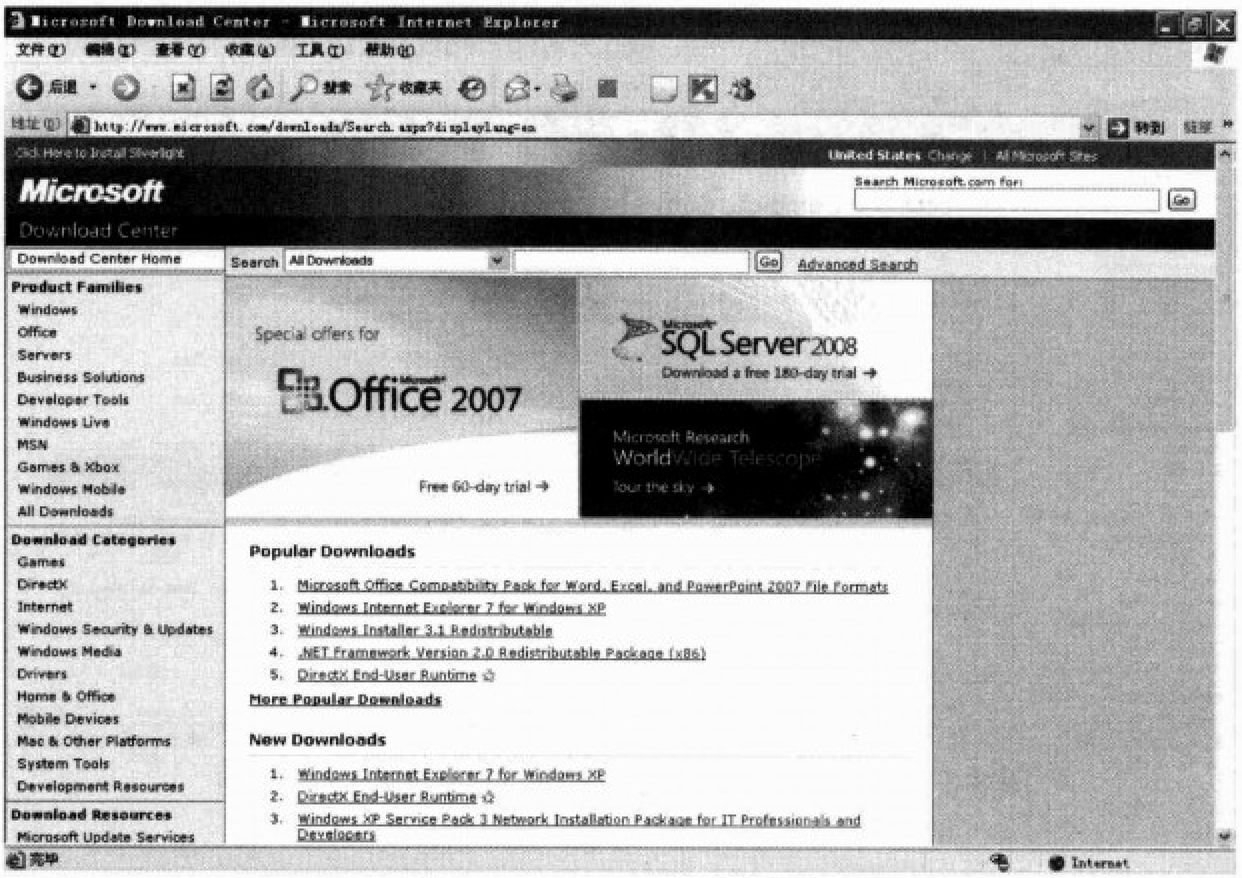


图 3-3 Microsoft 下载中心

STEP 03 在下载中心“Download Center”页面的左侧选择“DirectX”选项，如图 3-4 所示。

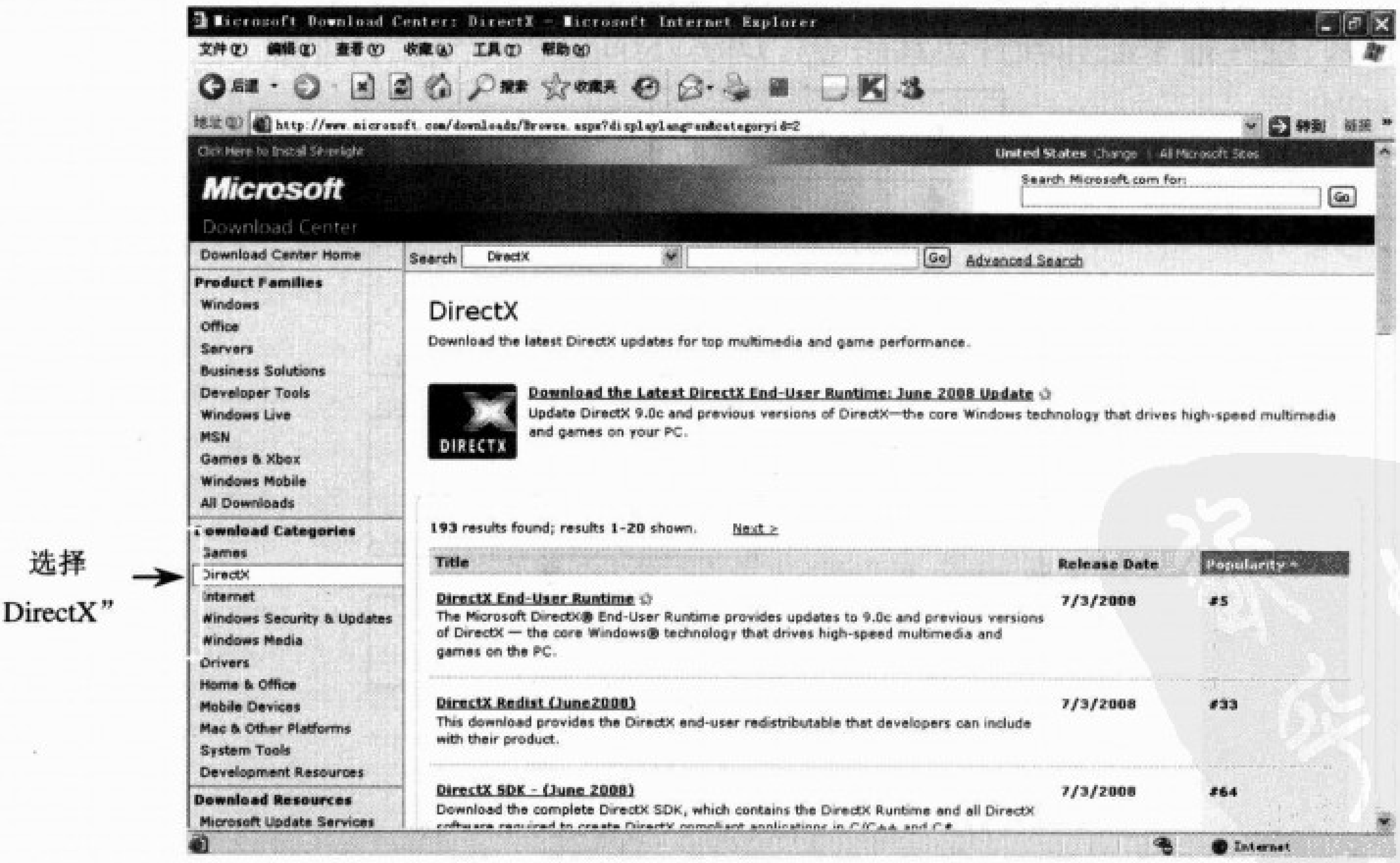


图 3-4 选择“DirectX”

STEP 04 在弹出的“DirectX”列表中单击“DirectX 9.0 Software Development Kit”超链接，弹出的页面如图 3-5 所示。

STEP 05 在弹出的页面中，单击“Countine”按钮，进入到下载页面，然后下载即可。

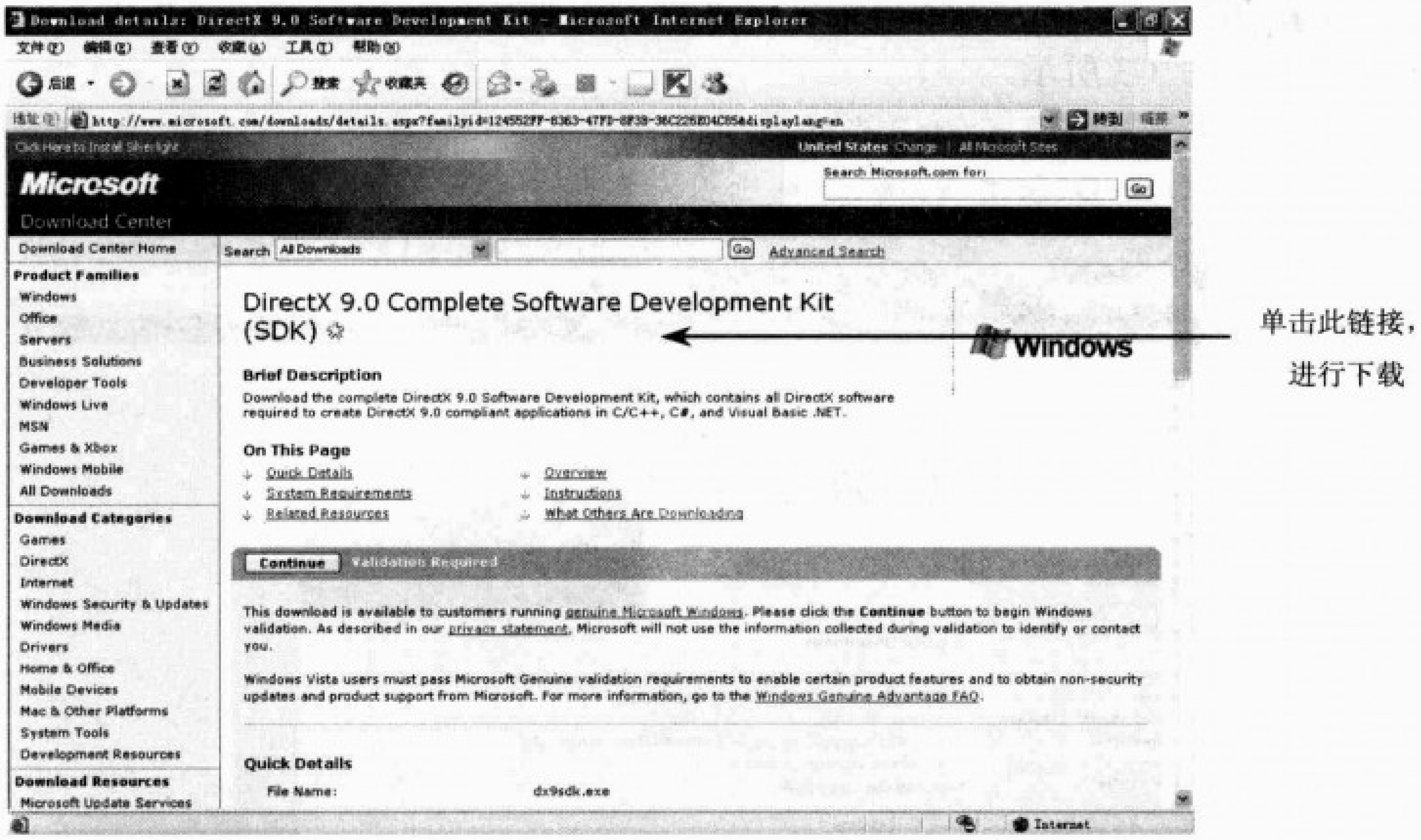


图 3-5 “DirectX 9.0 Software Development Kit”页面

3.1.4 DirectSound SDK的基本结构

DirectSound 的结构模块通常包括播放、回放（Playback）、三维音效（3D Sound）、音频捕获（Capture）、硬件抽象层和硬件仿真层等，DirectSound 的体系结构如图 3-6 所示。

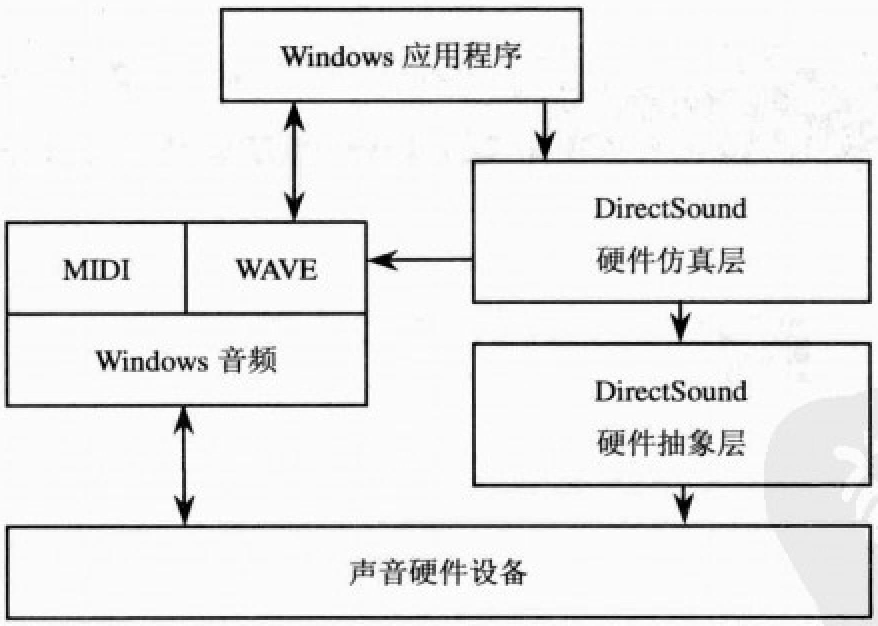


图 3-6 DirectSound 体系结构

1）音频的播放

DirectSound 缓冲区对象表示一个包含音频数据的缓冲区，这些数据以 PCM 格式被存储。该对象不仅可以用于开始、停止和暂停音频的播放，还能够设置音频数据中频率和格式等属性。缓冲区包括主缓冲区和次缓冲区。主缓冲区中是听者将要听到的音频信号，一般是将次缓冲区中信号混音后的结果，而次缓冲区中存放着许多单独的音频信号，有的可以直接播放，有的要混音，有的循环播放。主缓冲区的 DirectSound 自动创建，而次缓冲区需要应用程序

来创建。DirectSound 将次缓冲区中的音频混合后存入主缓冲区，再输出到相应的播放设备。另外，DirectSound 中没有解析音频文件的功能，因此需要读者在应用程序中将不同格式的音频信号改变过来。通过 DirectSoundNotify 接口，当播放到了一个用户指定的地方或者播放结束时，DirectSound 将动态地通知用户这一事件。

2）DirectSound 回放

DirectSound 缓存用于控制回放声音的开始、停止、暂停，回放声音时声音数据存放在 DirectSound 缓存中。DirectSound 将声音缓冲区分为主缓冲区（Primary Sound Buffer）和次缓冲区（Secondary Sound Buffer）。主缓冲区只有一个，直接与硬件相关，用于存放播放的数据。次缓冲区可以有多个，一个次缓冲区存放一路播放数据，次缓冲区的数据只有在主缓冲区上才能播放。一个次声音缓冲区对象代表一个声源，回放时将要播放的声音数据混合到主缓冲区中，由主缓冲区送到音频输出设备，这样就可以实现混音。在创建 DirectSound 时即创建了主缓冲区，应用程序需要设置其数据格式，次缓冲区由应用程序自己创建，应用程序只需要把数据放入次缓冲区就可以播放了，如图 3-7 所示。

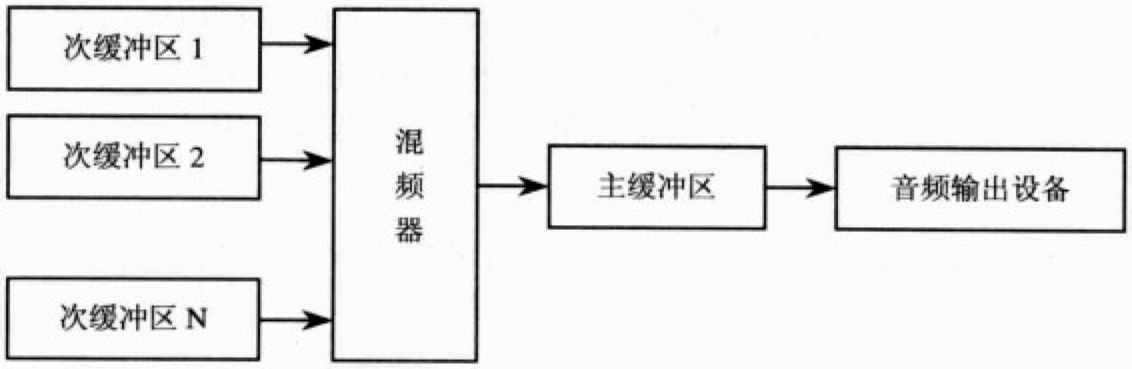


图 3-7 DirectSound 声音混音和回放图

通过 DirectSound 的 IDirectSoundNotify 组件接口，DirectSound 可以支持流方式播放，应用程序可以通过 IDirectSoundNotify 在次缓冲区中设置若干通知点，当播放指针到达通知点时就发出到达事件，应用程序就可以填充一段新的数据到次缓冲区中。

3）3D 音效

DirectSound 可以创建一个与 3D 图像结合的 3D 世界。应用程序指定声源和 3D 听众的空间位置，可以让移动的声音产生多普勒效应和控制一些声音随距离衰减速度等参数。通过三维坐标来表示声源的方向、位置、传播速度及 3D 听众的位置；3D 听众通过影响耳朵的几个声学因素来感知声音的位置；可以通过 DirectSound 3D 缓冲区来获得具有震撼力的 3D 音效。

4）DirectSound 捕获

DirectSound 对象可以查询音频捕获设备的性能，并为从输入源到捕获音频而创建缓冲区。其实，在 Windows 中早就有了包含音频的功能，而 DirectSoundCapture 与之比较其实并没有什么新的功能。不过，DirectSoundCapture API 使我们能够编写使用相同接口的音频捕获和播放程序，这也为将来可能出现的 API 改进提供了原始模型。此外，DirectSoundCapture 还能够捕获压缩格式的音频。

DirectSoundCaptureBuffer 对象表示一个用于捕捉音频数据的缓冲区，它可以循环利用，也就是说，当输入指针达到缓冲区的最后时，它会回到开始的地方。DirectSoundCaptureBuffer 对象的各种方式使读者能够设定缓冲区的属性、开始或停止操作、锁定某部分存储器，这样就可以安全地将这些数据保存或用于其他目的。

通过 DirectSoundNotify 接口，当捕捉到了一个用户指定的地方，或者捕捉结束时，DirectSound 将动态地通知用户这一事件。

5）硬件抽象层（Hardware Abstraction Layer）

DirectSound 通过硬件抽象层来访问声音硬件。硬件抽象层由 DirectSound 设备驱动程序控制，提供一个与声音硬件一致的接口。DirectSound 驱动程序实际上就有硬件抽象层扩展的 Windows 设备驱动程序。DirectSound 硬件抽象层描述声音硬件的性能，获取并释放声音硬件的控制。

大家网 TopSage.com

3.1.5 DirectSound SDK安装

为了能够使用 Visual C++进行 DirectSound 编程，必须有一套 DirectX 的 SDK，它包括了编译 DirectX 应用程序所需要的编译头文件（*.h）、编译库文件（*.lib）、示例，还有帮助。

要让 DirectX 应用程序能在脱离 VC 的环境下执行，用户的电脑还必须安装有 DirectX 运行库（Runtime Library），这也是常说的 DirectX 引擎（Engine）或驱动程序（Device Driver）。Microsoft 为了推行 DirectX 标准，将这套驱动程序库免费奉送，用户可以从 Microsoft 网站上下载。

上一节已经从 Microsoft 网站上获取了 DirectX 9.0 SDK，我们可以安装该程序了，具体的安装步骤与常规软件类似，就不赘述了。安装完毕后，可以在 SDK 下看到以下目录。

- SDKDev：软件开发工具包。
- Include：头文件。
- Lib：库文件。其中*.lib 用于 VC++编译器；*.lbw 用于 Watcom 编译器。
- Samples：示例源代码。
- Bin：例程的可执行文件。
- Doc：帮助和 Readme 文件。

这就表示一个完整的 SDK 已经安装到用户的电脑了。

3.1.6 选择最适合用户的DirectSound SDK

学习 DirectSound 开发通常有以下几种方法：查看帮助文档、查看 SDK 的示例代码和查看 SDK 基类源代码。帮助文档可以使读者在总体概念上有更深的理解，同时在帮助文档中包含了所有要用到的函数，并给出了详细的解释和使用方法。示例代码和基类代码可以同时进行，可以更直观地让读者从示例代码中来理解用 DirectSound 编程的步骤、流程及其所需功能实现的机理。

在帮助文档中，DirectSound SDK 给读者提供了一个英文版的帮助文件，它详细地说明了 DirectSound 在音频程序设计中的开发步骤、设计流程、函数接口代码等。DirectSound 可以播放 WAVE 声音文件，DirectSound 中的 3D 声音可以创造出虚拟的 3D 效果，而 DirectSound 中的一些标准的音效如回声、齐声等，则可以通过调用各个音效所对应的接口函数来实现。

读者可以在 DirectSound 程序设计中，根据自己的需要来选择所需的功能。在帮助文档中可以通过“目录”和“索引”立刻查找到所需的解释和函数。“目录”和“索引”如图 3-8 和图 3-9 所示。

图 3-8 DirectSound 帮助文档的“目录”

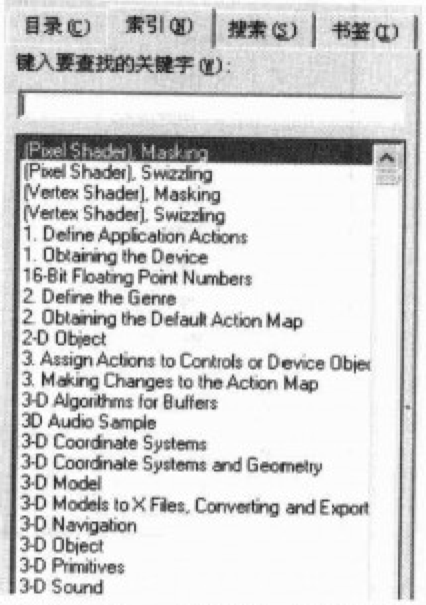


图 3-9 DirectSound 帮助文档中的“索引”

示例代码和基类代码是 DirectSound SDK 给读者的最直接、最直观的编程范例。在 Microsoft DirectX 9.0 SDK 中有很丰富的示例。有 Adjust Sound、Capture Sound、Enum Devices、Full Duplex Filter、Play 3D Sound、PlaySound、StreamData、VoiceManagement 等。同时 DirectSound SDK 包含一个 DirectX Samples Browser 应用程序，如图 3-10 所示。它集成了这些示例及示例代码，读者可以选择并加以演示，例如图 3-11 是特殊音效的示例。可以结合示例与自己开发的应用程序，找出自己开发的应用程序存在的问题，并加以修正，因此，在程序开发过程中，读者可以根据自己的需要来查看相应的示例。可以说这些示例非常实用，对读者提高编写程序成功率和缩短开发周期都能起到很好的作用。

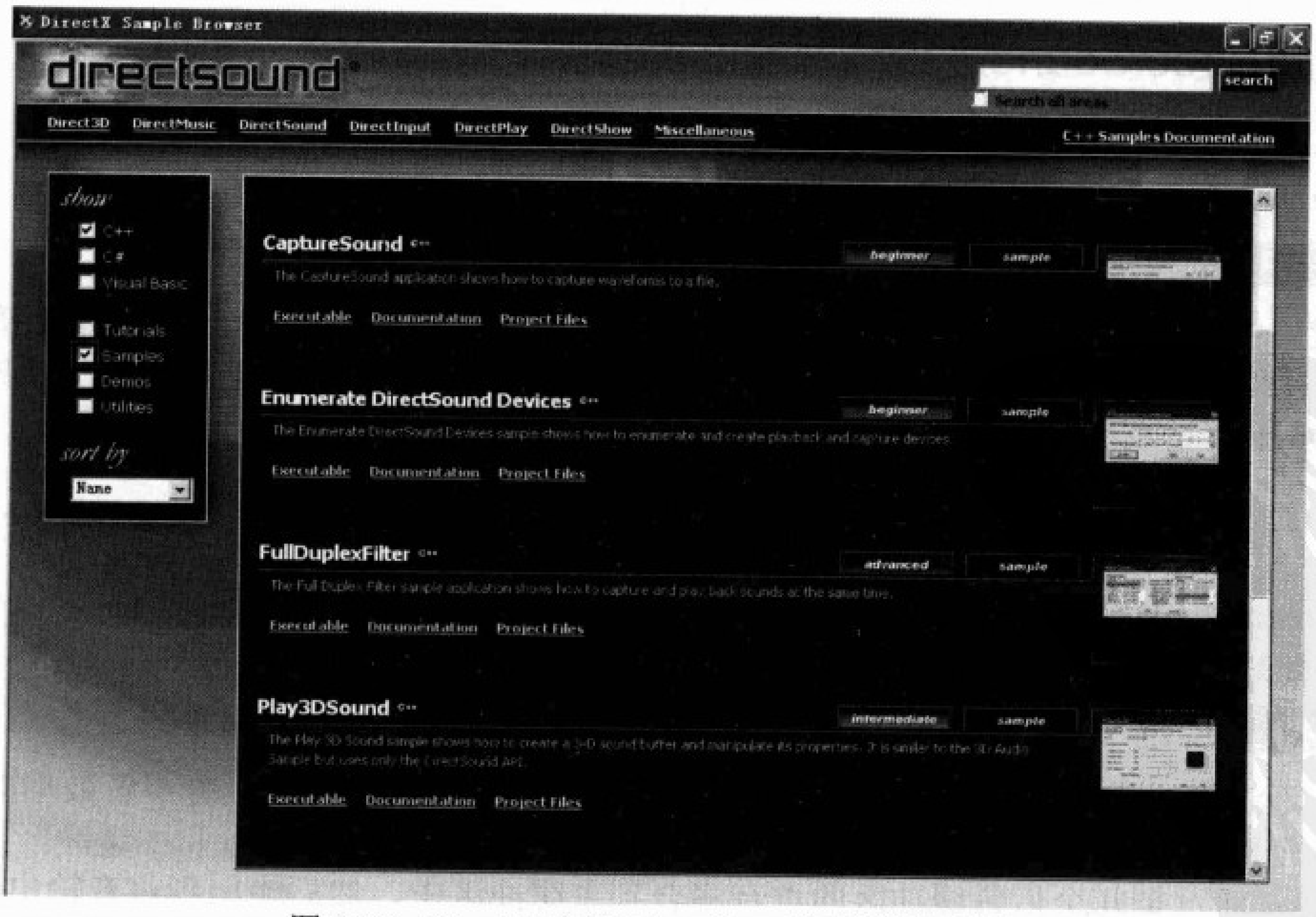


图 3-10 DirectX 中的 DirectSound 示例演示界面

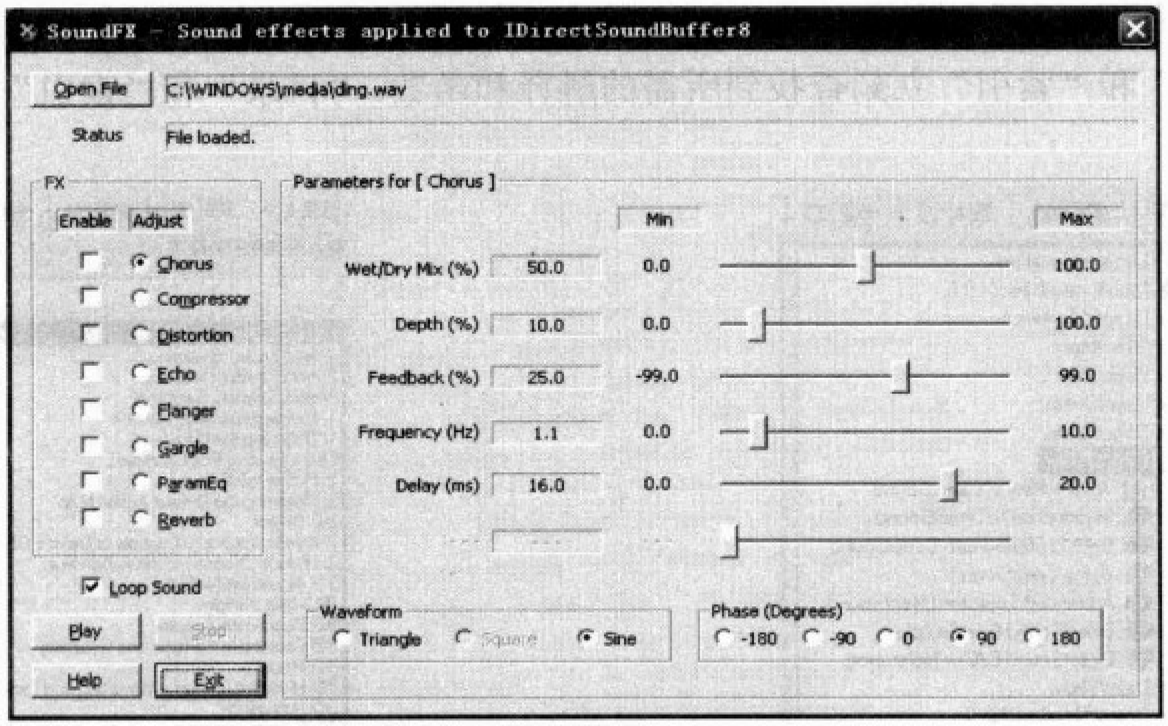


图 3-11　特殊音效示例

在 DirectSound SDK 中不仅包含了英文帮助文档和开发实例及示例代码，还有读者在开发 DirectSound 应用程序中必须加载到开发工具 Visual C++中的头文件和库文件，在这些头文件和库文件中包含了很多在开发应用程序过程中要用到的函数，所以在开发前要对开发工具进行环境配置，即要添加头文件和库文件，下节将具体介绍开发环境的配置方法和步骤。读者可以根据自己的需要进行添加并配置。

总之，在 DirectSound 开发中首先要很好地理解 DirectSound SDK 的结构和开发的基本流程，上面已经详细叙述过。然后根据需要选择合适的 DirectSound SDK，从而便于 DirectSound 应用程序的开发。

3.1.7　用DirectSound进行编程的必要步骤

下面简要介绍一下用 DirectSound 进行编程的几个必要步骤。

STEP 01 创建一个设备对象。

在代码中可以通过调用 DirectSoundCreat8 函数来创建一个支持 IDirectSound8 接口的对象，这个对象通常代表默认的播放设备。当然读者可以枚举可用的设备，然后将设备的 GUID 传递给 DirectSoundCreat8 函数。

注意：

DirectSound 虽然是基于 COM 的，但是并不需要初始化 COM 库，这些 DirectSound 都已经做好了。当然，如果用户使用 DMOs 特技，就要自己初始化 COM 库了。

STEP 02 创建一个次缓冲区，也叫后备缓冲区。

可以通过调用 IDirectSound8::CreateSoundBuffer 函数来创建一个包含声音数据的 Buffer 对象，这个对象主要用来获取处理数据，这种 Buffer 称做次缓冲区，以和主缓冲区区别开，DirectSound 通过把几个后备缓冲区的声音混合到主缓冲区中，然后输出到声音输出设备上，达到混音的效果。

STEP 03 获取 PCM 的数据类型。将 WAV 文件或者其他资源的数据读取到缓冲区中。

STEP 04 把数据读取（加载）到缓冲区。

可以通过调用 IDirectSoundBuffer8::Lock 函数来为写入此缓冲区做准备。通常这个函数返回一个表示缓冲区的内存地址，把数据从私有的缓冲区中复制到这个地址中，然后调用 IDirectSoundBuffer8::Unlock。

STEP 05 播放缓冲区中的数据。

通过调用 IDirectSoundBuffer8::Play 来播放缓冲区中的音频数据，也可以通过 IDirectSoundBuffer8::Stop 来暂停播放数据，可以反复地停止或者播放音频数据。一次只能播放缓冲区中的一个声音实例，如果同时创建了多个缓冲区，那么就可以同一时间来播放这些数据，这些声音会自动进行混音。至于同时可以播放几个辅助缓冲区则由硬件设备的性能决定。

但是这些都是针对最简单的应用而采用的步骤，即一个很短的声音刚好放在一个缓冲区里面。如果播放声音比较长，那么这些声音数据就会变成声音流，就是缓冲区中播放的数据会周期地被新数据代替。

3.2 DirectSound的详细使用方法

上面对 DirectSound 的编程步骤进行了简单介绍，本节将详细介绍 DirectSound 的使用方法。表 3-2 中简单介绍了 DirectSound 开发中常用的对象。

表 3-2 DirectSound 中播放声音的常见对象

对象	数量	作用	主要接口
设备对象	每个应用程序只有一个设备对象	用来管理设备，创建声音缓冲区	IDirectSound8，
次缓冲区对象	每一个声音对应一个次缓冲区	用来管理一个静态的或者声音流，然后在主缓冲区中播放	IDirectSoundBuffer8，IDirectSound3DBuffer8，IDirectSoundNotify8
主缓冲区对象	一个应用程序只有一个主缓冲区	将次缓冲区的数据进行混音，并且控制 3D 音效的参数	IdirectSoundBuffer，IDirectSound3DListener8
音效对象	每个次缓冲区有 0 个或多个	对次缓冲区的声音数据进行转换	8 个特技接口 IDirectSoundFXChorus8

在应用中运行 DirectSound 的第一步就是要创建一个设备对象，它代表播放声音的设备。然后用这个对象来创建缓冲区对象。

次缓冲区被创建后通过应用程序来管理。DirectSound 自动地创建和管理主缓冲区，一个应用程序可以在没有获得对象接口时播放声音，但是，为了获得 IDirectSound3DListener8 接口，应用程序必须专门创建一个主缓冲区对象。

注意：

在 Windows 驱动模型的情况下，通过内部的混频器来操作，主缓冲区并不真正地包含任何数据。为了得到更多的信息，请参考 DirectSound 驱动模型。

一个短的声音可以完全被加载到次缓冲区中，通过调用一个简单的函数就可以在任何时候播放它。长的声音不得不变成流的形式，应用程序可以用两种方法来确定何时把更多的数据流加载到次缓冲区：一种是通过拖动播放指针指向的位置，另外一种是当播放指针指向某些位置时发出通知。

次缓冲区包括音效，如回声和齐声等，它们还包括 3D 控制能力。这些 3D 参数用主缓冲区对象的 IDirectSound3DListener8 接口来控制。

3.2.1 DirectSound设备对象

DirectSound 设备对象通常代表一个声音设备，它负责管理设备和创建缓冲区。

多个应用程序能为多个声音设备创建设备对象。当输入焦点在应用程序间改变时，音频输出会自动从一个应用程序转换到另一个应用程序。结果，当输入焦点改变时，应用程序不能重复播放缓冲区或者停止缓冲区的数据。

1. 枚举声音设备

如果应用程序使用用户默认的播放设备来播放声音，那么就没有必要枚举所有的输出设备了。如果是通过 DirectSoundCreat8 函数来创建一个设备对象，那么就要给这个对象指定一个默认的设备，当然如果遇到下面的这些情形，用户就要枚举设备对象了。

- 应用程序并不支持所有的播放设备。
- 应用程序需要两个或者两个以上的设备。
- 用户自己选择播放设备。

要枚举 DirectSound 设备，首先要定义一个回调函数，这个回调函数可以被系统中的每个设备调用。在这个回调函数里面可以做任何事情，命名也没有任何限制，但是这个函数必须以 DSEnumCallback 为原型。如果还想枚举，则这个回调函数就返回 TRUE；如果枚举结束，例如找到了所需要的设备，那么就返回 FALSE。

下面举一个回调函数的例子，它把每一个枚举出来的设备加入到一个列表框（Combox）中，将相应的 GUID 存储到一个项（Item）中，这个函数开头的 3 个参数的值由设备的驱动程序提供，第 4 个参数由 DirectSoundEnumerate 函数提供，这个参数用来传递一个 32 位值。本例是列表框的窗口句柄。

```
BOOL CALLBACK DSEnumProc(LPGUID lpGUID,  //回调函数定义
                         LPCTSTR lpszDesc,
                         LPCTSTR lpszDrvName,
                         LPVOID lpContext )
{
  HWND hCombo = (HWND)lpContext;
```

```
    LPGUID lpTemp = NULL;
     if (lpGUID != NULL)        //当主声音设备为NULL时
    {
      if ((lpTemp = (LPGUID)malloc(sizeof(GUID))) == NULL)
      {
            return(TRUE);
      }
  memcpy(lpTemp, lpGUID, sizeof(GUID)); //把lpGUID所指的区域复制到lpTemp
    }
     ComboBox_AddString(hCombo, lpszDesc);
    ComboBox_SetItemData(hCombo,
        ComboBox_FindString(hCombo, 0, lpszDesc),
        lpTemp );
    free(lpTemp);
    return(TRUE);
  }
  //枚举设备都是在包含列表框的对话框被初始化时才进行,
  if (FAILED(DirectSoundEnumerate((LPDSENUMCALLBACK)DSEnumProc,
  (VOID*)&hCombo)))
  {
  EndDialog(hDlg, TRUE);   //清除模式对话框，并使系统中止对该对话框的处理
  return(TRUE);
  }
```

本例中，列表框的句柄 hCombo 作为参数传递到 DirectSoundEnumerate()函数，然后这个函数又把它传给回调函数。这个参数可以是想传递的 32 位值。

2．创建设备对象

创建设备对象最简单的方法是调用 DirectSoundCreate8 函数，这个函数的第一个参数指定了和这个对象对应设备的 GUID，也可以通过传递下面的参数来指定对应的默认设备。

- DSDEVID_DefaultPlayback：默认系统的声音输出设备，参数也可以是 NULL。
- DSDEVID_DefaultVoicePlayback：默认声音输出设备，通常是一个二级设备，例如 USB、麦克风、耳机等。

如果没有对应的设备驱动程序，则调用 DirectSoundCreate8 会失败。如果没有声音输出设备，则此函数就返回 Error。在 VXD 驱动程序下或者声音设备被另一个使用标准的波形（Waveform）格式的 API 所控制，该函数也会返回 Error。

下面的代码是创建一个对应与默认设备对象，同时得到一个 IDirectSound8 接口。

```
LPDIRECTSOUND8 lpds;
HRESULT hr = DirectSoundCreate8(NULL, &lpds, NULL));
```

如果是调用 COM 来创建设备对象，则可以使用以下代码。

```
HRESULT hr = CoInitializeEx(NULL, 0);
if (FAILED(hr))
{
```

```
ErrorHandler(hr);           //增加错误处理.
}
LPDIRECTSOUND8 lpds;
hr = CoCreateInstance(&CLSID_DirectSound8,NULL,  //创建一个COM对象
CLSCTX_INPROC_SERVER,
IID_IDirectSound8,
(LPVOID*) &lpds
);
if (FAILED(hr))
{
ErrorHandler(hr);           //增加错误处理
}
hr = lpds->Initialize(NULL);
if (FAILED(hr))
{
ErrorHandler(hr);           //增加错误处理
}
CoUninitialize();
```

此代码用在调用 COM 设备对象的场合，如果程序中要用到声音特效，那么就必须在创建对象前显示初始化 COM。DirectSoundCreate 函数不会使 CoInitialize 或者 CoInitializeEx 被调用。

3. 设置设备的协作级别

Windows 是一个多任务的操作系统，同时可以有多个程序使用同一个设备。通过协作级别的设定，DirectX 可以使每个应用程序在访问设备的时候不发生冲突。每一个 DirectSound8 接口都有一个协作级别，用来决定访问设备的程度。

当创建了设备对象之后，一定要调用 IDirectSound8 的 SetCooperativeLevel 设置协作级别，否则会听不到声音。

```
HRESULT hr = lpDirectSound->SetCooperativeLevel(hwnd, DSSCL_PRIORITY);
if (FAILED(hr))         //设置协作级别函数
{
ErrorHandler(hr);       //增加错误句柄     .
}
```

hwnd 是应用程序的窗口的句柄。DirectSound 为设备定义了 3 个协作级别：DSSCL_NORMAL、DSSCL_PRIORITY 和 DSSCL_WRITEPRIMARY。

在普通协作级别（DSSCL_NORMAL）下，应用程序不能设定主缓冲区的格式，也不能对主缓冲区进行写操作，所有这个级别的应用程序都使用 22kHz、立体声、采样精度 8 位的主缓冲区，所以各个应用程序可以任意切换。

在优先协作级别（DSSCL_PRIORITY）下，DirectSound 设备设定为此级别，则应用程序可以优先使用硬件资源，例如使用硬件进行混音。并且可以设定主缓冲区的格式，游戏程序应采用此协作级别，还可以调整设备内存的分布。

在写主缓冲区协作级别（DSSCL_WRITEPRIMARY）下，当设定为此级别时，应用程序可

以直接访问主缓冲区。在这个模式下，应用程序必须直接操作主缓冲区。如果读者想直接把音频数据写入到主缓冲区中，则应用程序的协作级别模式必须设定为此协作级别，如果读者的应用程序没有设置到这个层次，那么所有的对主缓冲区的 IDirectSoundBuffer::Lock 都会失败。

如果应用程序的协作级别设置为写主缓冲区协作级别，且应用程序所在窗口位于最前方，那么所有其他的应用程序的此缓冲区就会停止，并且标志为丢失。当此应用程序转到后面后，它的主缓冲区就会标志为丢失。当它再次回到最前时，必须恢复到它原来的状态。如果系统里面没有 DirectSound 设备，则无法设定写主缓冲区的协作级别。要判断系统中是否有 DirectSoound 设备，可以通过 IDirectSound8::GetCaps 方法来检查。

4. 设备特性

DirectSound 可以让应用程序检查声音设备的性能。然而许多程序都不需要这么做，因为 DirectSound 会自动利用硬件的一切可用的硬件加速，不需要手动选择。但是，高性能的应用程序能用这些信息来按比例分配可用的硬件。例如，如果硬件混音成为可能，则应用程序可以播放更多的声音；如果不能混音，则只能播放较少的音乐。

当调用 DirectSoundCreate8 函数来创建一个设备对象后，应用程序就可以通过 IDirectSound8::GetCaps 函数的方法来获取声音设备的性能属性，下面代码就描述了这个过程。

```
DSCAPS dscaps;
dscaps.dwSize = sizeof(DSCAPS);
HRESULT hr = lpDirectSound->GetCaps(&dscaps);      //获取声音设备的性能属性
if (FAILED(hr))
{
  ErrorHandler(hr);                                //增加错误处理.
}
```

通过 DSCAPS 结构，该函数可以接受声音设备的性能和资源信息，包括每种资源的最大数目和当前可以利用的资源。但是这个函数在调用之前，必须要初始化这个结构的 dwSize 成员。

如果应用程序按比例分配设备性能，那么在硬件设备上分配资源的时候，就要调用 IDirectSound8::GetCaps 来检查是否有足够多的资源满足要求。

5. 扬声器配置

DirectSound 用扬声器相对于听者的位置来配置声源，从而使声音系统达到最优的 3D 效果。

在 Windows 98、Windows 2000 及后来的操作系统中，用户可以通过控制面板来配置扬声器，应用程序可以通过 IDirectSound8::GetSpeakerConfig 函数来获取这些设置的值。应用程序不应该调用 IDirectSound8::SetSpeakerConfig 函数来设置扬声器，因为这些设置会影响到其他用户和应用程序。

3.2.2 DirectSound缓冲区对象

当存储和播放几个音频流的时候，应用程序要给每一个音频流创建一个次缓冲区对象。

DirectSound缓冲区对象控制波形数据从源头到目的地进行传送。数据源可能是一个合成器、另外的缓冲区、一个WAV文件或者是一个资源。对很多缓冲区来说，目的地是一个叫做主缓冲区的混音引擎。数据从主缓冲区进入到硬件并转变成简单的声音波形。

1. 缓冲区

一个应用程序必须至少创建一个次缓冲区来存储和播放单个的声音。次缓冲区和应用程序的生存期一样长，也可以在不需要的时候销毁。次缓冲区可以是一个包含单个短声音的静态缓冲区，也可以在播放时不断更新数据的流缓冲区。为了限制内存需求，播放较长的声音文件时应该使用流缓冲区，这些缓冲区里面只存几秒钟的播放数据。

可以通过同时播放几个次缓冲区中的声音来对它们进行混音。只要硬件设备的性能足够，就可以同时播放任意数目的次缓冲区。次缓冲区的创建方式不完全一样，特征如下。

- 格式：缓冲区的格式必须要与所播放的音频数据格式一致。
- 控制：不同的缓冲区有不同的控制方法，比如音量、频率，以及在不同二维或者三维空间中移动，在创建缓冲区时，应该指定需要的控制参数值，例如，在非3D环境下，就不要创建一个3D缓冲区。

2. 创建次缓冲区

可以调用IDirectSound8::CreateSoundBuffer函数来创建一个缓冲区对象。此函数返回一个IDirectSoundBuffer接口指针，通过这个接口，应用程序可以获取IDirectSound8接口。

下面的代码演示了如何创建一个次缓冲区，并返回一个IDirectSounBuffer8接口。

```
HRESULT  CreateBasicBuffer(LPDIRECTSOUND8
lpDirectSound, LPDIRECTSOUNDBUFFER8* ppDsb8)  //创建次缓冲区结构体
{
  WAVEFORMATEX wfx;
  DSBUFFERDESC dsbdesc;                            //创建结构体对象
  LPDIRECTSOUNDBUFFER pDsb = NULL;
  HRESULT hr;
  //创建WAV格式结构体
  memset(&wfx, 0, sizeof(WAVEFORMATEX));
  wfx.wFormatTag = WAVE_FORMAT_PCM;   //WAV文件应为WAVE_FORMAT_PCM
  wfx.nChannels = 2;                  //设置通道数
  wfx.nSamplesPerSec = 22050;         //设置采样率
  wfx.nBlockAlign = 4;                //设置块对齐大小
  wfx.nAvgBytesPerSec = wfx.nSamplesPerSec * wfx.nBlockAlign;
  wfx.wBitsPerSample = 16;            //设置位数为16
  //创建DSBUFFERDESC结构体
   memset(&dsbdesc, 0, sizeof(DSBUFFERDESC));
  dsbdesc.dwSize = sizeof(DSBUFFERDESC);
  dsbdesc.dwFlags =
 DSBCAPS_CTRLPAN | DSBCAPS_CTRLVOLUME | DSBCAPS_CTRLFREQUENCY;
  dsbdesc.dwBufferBytes = 3 * wfx.nAvgBytesPerSec;  //设置缓冲区长度为多少秒，这里
就设置为wfx.nAvgBytesPerSec乘以多少，这里统一设置成1秒
```

```
    dsbdesc.lpwfxFormat = &wfx;
    //创建缓冲区
    hr = lpDirectSound->CreateSoundBuffer(&dsbdesc, &pDsb, NULL);
    if (SUCCEEDED(hr))
    {  //取得IDirectSoundBuffer8缓冲区接口对象
       hr = pDsb->QueryInterface(IID_IDirectSoundBuffer8, (LPVOID*) ppDsb8);
       pDsb->Release();
    }
    return hr;
}
```

此例中创建了一个可以持续播放 3 秒钟的流数据缓冲区，非流缓冲区应该大到可以容纳整个音频数据。

如果缓冲区的位置没有指定，那么 DirectSound 在条件允许的情况下可以把它放到硬件控制的内存区域。因为硬件缓冲区的混音是通过声卡处理器进行的，这可以很大程度地减少应用程序的影响。

如果想要创建自己控制的缓冲区位置，就一定要将 DSBUFFERDESC 结构中的 dsbdesc.dwFlags 成员变量设置为 DSBCAPS_LOCHARDWAVE 或者 DSBCAPS_ LOCSOFTWARE。如果设定了 DSBCAPS_LOCHARDWAVE 标志而资源不足，则缓冲区创建失败。

如果要使用 DirectSound 中的管理声音的特性，那么在创建缓冲区时就要设定 DSBCAPS_LOCDEFER 标志，这个标志会使缓冲区的创建推迟到播放的时候才开始。

可以通过 IDirectSoundBuffer8::GetCaps 方法来检查缓冲区 dwFlags 的设置情况。缓冲区对象属于创建它的设备对象，当设备对象被销毁时，它所创建的全部缓冲区都会被销毁，无法再使用。

3. 复制缓冲区

用 IDirectSound8::DuplicateSoundBuffer 可以同时创建两个或者多个包含系统数据的次缓冲区，但是不允许复制主缓冲区。因为复制的缓冲区和原来的缓冲区共享内存，对原来缓冲区的改变会反映到复制缓冲区中。

4. 缓冲区控制属性

当创建缓冲区的时候，应用程序必须设置缓冲区的控制属性。这些属性设置在 DSBUFFERDESC 结构的 dwFlags 成员中，它可以同时包含表中的一个或多个取值。创建缓冲区用到的标志位如表 3-3 所示。

表 3-3 创建缓冲区用到的标志位

标志	描述
DSBCAPS_CTRL3D	声音可以在 3 个方向上移动
DSBCAPS_CTRLFX	可以在缓冲区添加音效
DSBCAPS_CTRLFREQUENCY	可以改变声音的频率

（续表）

标志	描述
DSBCAPS_CTRLPAN	声音可以从左声道移动到右声道
DSBCAPS_CTRLPOSITIONNOTIFY	可以在缓冲区中设置通知的位置
DSBCPAS_CTRLVOLUME	可以改变音量的大小

注意：

有些标志位的组合是不允许的，具体信息可以参考 DSBUFFERDESC 结构。

为了使声卡达到更好的性能，最好改变应用程序会用到的控制属性，其他的属性最好采用 DirectSound 默认的设置。

DirectSound 通过控制属性来决定是否可以在硬件设备上分配资源给缓冲区。例如，一个设备可能支持硬件缓冲区，但是不支持面板控制，此时，当 DSBCAPS_CTRLPAN 标志被设置的时候，DirectSound 就会使用硬件加速。

如果应用程序想调用缓冲区不支持的控制属性，就会调用失败。例如，如果想通过 IDirectSoundBuffer8::SetVolume 方法来改变音量，那么只有在创建缓冲区时设定了 DSBCAPS_ CTRLVOLUME，这个函数才可以调用成功，否则就会调用失败。

5. 缓冲区的 3D 算法

当创建的次缓冲区的控制标志位被设定为 DSBCAPS_CTRL3D 时，可以同时给创建的缓冲区指定一个 3D 算法。在默认的情况下，不使用 HRTF，只通过面板和音量控制来决定声音对听者的位置。具体的 3D 算法将在第 5 章做详细介绍。

6. 填充和播放静态缓冲区

包含整个音频数据的缓冲区称为静态缓冲区。尽管它可以用来播放其他声音，但是通常对静态缓冲区只写入一次。

静态缓冲区的创建和管理与流缓冲区相似。唯一的区别就是使用方式不同，静态缓冲区只填充一次就开始播放，而流缓冲区则是边播放边填充数据，更新数据。给缓冲区加载数据有以下 3 个步骤。

STEP 01 调用 IDirectSoundBuffer8::Lock 函数来锁定所有缓冲区。要指定锁定缓冲区中开始写入的数据偏移量，并且得到返回哪个位置的内存地址。

STEP 02 用标准的数据复制方法，将音频数据复制到返回的那个地址上。

STEP 03 调用 IDirectSoundBuffer8::Unlock 解锁该地址。

这些步骤可以通过下例来演示，lpdsbStatic 是 IDirectSoundBuffer8 接口指针。

```
LPVOID lpvWrite;
DWORD dwLength;
if (DS_OK == lpdsbStatic->Lock(
0,                        //开始锁存位置的偏移量
0,                        //锁存的大小
```

```
&lpvWrite,                    //获得锁存的起始地址
&dwLength,                    //获得锁存区域的大小
NULL,                         //不需要循环覆盖的地址
NULL,                         //不需要循环覆盖的大小
DSBLOCK_ENTIREBUFFER))        //标志位
{
memcpy(lpvWrite, pbData, dwLength);
lpdsbStatic->Unlock(          //调用解锁函数
lpvWrite,                     //锁存开始的地址
dwLength,                     //锁存的大小
NULL,                         //非覆盖部分.
0);                           //非覆盖大小.
}
else
{
ErrorHandler();               // 增加错误处理
}
```

将数据加载到缓冲区中就可以播放，调用 IDirectSoundBuffer8::Play 方法。

```
lpdsbStatic->SetCurrentPosition(0);
HRESULT hr = lpdsbStatic->Play(
0,                  //未使用
0,                  //声音管理优先级
0);                 //标志位
if (FAILED(hr))
{
ErrorHandler();     //增加错误处理
}
```

此例中没有设置 DSBPLAY_LOOPING 标志，当缓冲区播放到最后时就会自动停止，也可以调用 IDirectSoundBuffer8::Stop 方法来停止播放。当手动结束时，播放指针就停留在所播放的位置。所以，例子中调用 IDirectSoundBuffer8::SetCurrentPosition 函数来设定播放指针是为了保证从头开始播放。

7. 使用流缓冲区

流缓冲区用来播放一些较长的声音。因为数据比较长，无法一次性完全放入缓冲区，所以一边播放一边将新的数据放入缓冲区中。我们可以调用 IDirectSoundBuffer8::Play 函数来播放流缓冲区。

注意：

在 dwFlags 参数中设定 DSBPLAY_LOOPING 标志。

我们通过调用 IDirectSoundBuffer8::Stop 方法来中断播放，该方法会立即停止缓冲区播放。要确保所有的数据都被播放，就要将播放位置设置到缓冲区结尾位置或通知位置。将音频流放入到一个缓冲区需要以下 4 个步骤。

STEP 01 确保缓冲区已经做好接收新数据的准备，可以拖放指针或者等待通知。

STEP 02 调用 IDirectSoundBuffer8::Lock 函数来锁定缓冲区的位置，此函数返回一个或两个可以写入数据的地址。

STEP 03 使用标准的复制方法把声音数据写入缓冲区地址中。

STEP 04 用 IDirectSoundBuffer::Unlock 函数来解除对缓冲区的锁定。

IDirectSoundBuffer8::Lock 可能返回两个地址的原因在于锁定小于缓冲区的任意大小的内存，而不去考虑锁定块的起始位置。

例如，假设锁定了 30000 字节，从偏移 20000 开始，如果缓冲区的大小为 40000 字节，那么此时就会返回以下 4 个数值。

- 内存地址的偏移位置 20000。
- 从偏移位置到缓冲区的最尾部的字节数，也是 20000 字节。应该从第一个地址开始写入这么多数据。
- 偏移量为 0 的内存地址。
- 从起始点开始的字节数，也就是 10000 字节，写数据时要从第二个地址开始写入这么多数据。

如果不需要从缓冲区头开始锁定，则最后返回的两个数值都为 NULL。

当然可以锁定整个缓冲区，但是在播放的时候不能这么做，通常只是更新所有缓冲区的一部分。例如，在播放指针到达了缓冲区的 1/2 位置时，就要将前 1/4 的数据更新，依次类推。从播放指针到写指针之间的数据是不能被写的。下面的函数可以实现从 dwOffset 开始向缓冲区写数据。

```
BOOL AppWriteDataToBuffer(LPDIRECTSOUNDBUFFER8 lpDsb,  //缓冲区指针
DWORD dwOffset,            //写指针
LPBYTE lpbSoundData,       //写数据的开始位置
DWORD dwSoundBytes)        //要写的数据块的大小
{
LPVOID lpvPtr1;
DWORD dwBytes1;
LPVOID lpvPtr2;
DWORD dwBytes2;
HRESULT hr;
//获得写数据块的内存地址，可能是两个地址
hr = lpDsb->Lock(dwOffset, dwSoundBytes, &lpvPtr1,
&dwBytes1, &lpvPtr2, &dwBytes2, 0);
//如果缓冲区丢失，则重新存储恢复并锁存
if (DSERR_BUFFERLOST == hr)
{
lpDsb->Restore();  //重新恢复
hr = lpDsb->Lock(dwOffset, dwSoundBytes, //调用锁存函数
&lpvPtr1, &dwBytes1,
&lpvPtr2, &dwBytes2, 0);
}
if (SUCCEEDED(hr))
```

```
{
//写指针
CopyMemory(lpvPtr1, lpbSoundData, dwBytes1);
if (NULL != lpvPtr2)
{
CopyMemory(lpvPtr2, lpbSoundData+dwBytes1, dwBytes2);
}
//解除锁存
hr = lpDsb->Unlock(lpvPtr1, dwBytes1, lpvPtr2,dwBytes2);
if (SUCCEEDED(hr))
{
//成功
return TRUE;
}
}
//锁存，解锁或者存储失败
return FALSE;
}
```

上面的程序实现了从 dwOffset 开始向声音缓冲区写数据。首先定义了写数据到缓冲区的函数，然后获取要写的数据块的内存地址并锁存，向缓冲区写数据，最后解除锁存，则成功地把数据写入声音缓冲区中。

8. 重放控制

应用程序通过调用 IDirectSoundBuffer8::GetVolume 和 IDirectSoundBuffer8::SetVolume 函数来获取或者设置正在播放的音频的音量大小。如果设置主缓冲区的音量，则会改变声卡的音量。音量的大小用 100dB 衰减来度量，一般不能增强默认的音量。

注意：

分贝的衰减不是线性的，减少 3dB，能量就减少一半，最大值衰减 100dB 就听不到了。

通过调用 IDirectSoundBuffer8::GetFrequency 和 IDirectSoundBuffer8::SetFrequency 方法可以获取设置声音播放的频率，而主缓冲区的频率不能改变。

通过 IDirectSoundBuffer8::GetPan 和 IDirectSoundBuffer8::SetPan 函数可以设置声源在左右声道播放的位置，具有 3D 特性的缓冲区不能调用该函数。

IDirectSoundBuffer8::GetPan 函数原型为：

```
HRESULT GetPan(
 LPLONG plPan  //获取扬声器左右声变量的地址
);
```

此函数获取扬声器的左右声道播放的位置。

IDirectSoundBuffer8::SetPan 函数的原型为：

```
HRESULT GetPan(
 LPLONG plPan  //控制扬声器左右声变量的地址
);
```

此函数设置扬声器的左右声道播放的位置。

9. 混音

如果次缓冲区的声音同时播放，则会自动地在主缓冲区里混音。在 Windows 驱动模式（WDM）下，混音在核心混音器中完成。不同的次缓冲区可以有不同的 WAV 格式（比如，不同的采用频率），在混音时，所有的次缓冲区都会转换成主缓冲区或者核心混音器的格式。

在虚拟设备驱动模式（VXD）下，如果次缓冲区都采用相同的 WAV 格式，并且硬件的输出格式和声音格式匹配，则混音器不做任何的转换，会产生最好的音质。应用程序可以通过 IDirectSoundBuffer8::SetFormat 函数来设置硬件的输出格式，注意一定要在优先协作级别下才能做到，并且，一定要在创建次缓冲区之前调用这个函数。DirectSound 每次都会把上次设置的格式保存下来。

在 WDM 模式下，对主缓冲区的设置没有作用，因为格式是由内核混音器决定的。

10. 循环播放

DirectSound 不支持缓冲区内的循环播放或循环播放一部分声音。虽然可以通过在播放到缓冲区尾时把播放指针移动到缓冲区头来循环播放，但会导致短暂停顿，因为 DirectSound 会丢弃原来预处理的数据，而会重新进行预处理。因此，如果要重复循环播放，一定要在流缓冲区中播放并定时更新缓冲区中的数据。

11. 缓冲区管理

通过 IDirectSoundBuffer8::GetCaps 方法可以获取 DirectSoundBuffer 对象的一些属性。应用程序可以通过 IDirectSoundBuffer8::GetStatus 方法来判断缓冲区是正在播放还是处于停止状态。通过 IDirectSoundBuffer8::GetFormat 方法可以获取 buffer 中音频数据的格式信息，也可以通过调用 IDirectSoundBuffer8::SetFormat 方法来设置主缓冲区的数据格式。

在某些条件下，声音缓冲区的数据可能会丢失，例如，缓冲区位于声卡的内存中而另一个应用程序控制了硬件资源。当具有写主缓冲区协作级别的应用程序移动到前端时，DirectSound 会使其他的缓冲区丢失以便可以直接写入主缓冲区。

如果对已经丢失的缓冲区进行操作，则调用 IDirectSoundBuffer8::Lock 或者 IDirectSoundBuffer8::Play 方法就会返回一个 DSERR_BUFFERLOST 错误码。当造成缓冲区丢失的应用程序协作级别低于写主缓冲区协作级别或者移动到后端时，其他的应用程序可以通过调用 IDirectSoundBuffer8::Restore 方法来重新分配缓冲区内存。如果分配成功，则这个方法就会恢复缓冲区中的内容并保留一切设置，比如音量、面板等。但是，恢复的缓冲区中不包含合法的数据，因此，应用程序要重新向该缓冲区中写入数据。

3.2.3 使用WAV数据

在 WDM 模式下，DirectSound 可以播放任何格式的声音数据，只要这些数据可以用 WAVEFORMATEXTENSIBLE 或者 WAVEFORMATEX 结构并且硬件也支持这种格式。而软

件缓冲区和非 WDM 硬件缓冲区只支持 8 位和 16 位未被压缩的格式。

波形数据一般保存在文件或资源交换文件格式（RIFF）的资源中。这些数据包括对 WAV 格式的描述，包括一些参数，比如采样频率和输出通道的数量等参数。

DirectSound 缓冲区支持如下 WAV 格式：多声道 WAV 格式、读取 WAV 数据、计算 WAV 声音的持续事件。

1）多声道 WAV 格式

在 WDM 模式下，DirectSound 缓冲区支持 WAV 格式，为了向不同的扬声器输出不同的声音。DirectSound 不支持对多声道格式的缓冲区进行声音特效和 3D 处理。

如果设置了 DSBCAPS_CTRL3D 或者 DSBCAPS_CTRLFX 标志，那么创建多声道格式缓冲区就会失败。

2）读取 WAV 数据

WAV 文件是资源交换文件格式（RIFF）的，它由可变数据的数据库组成，这些数据块或者包含头信息（声音格式），或者包含数据（声音本身）。Win32 API 提供了打开和关闭 RIFF 文件、寻找数据块等操作的函数。通常这些函数都以“mmio”开头。

在应用程序中要保存声音数据，就要把 WAV 文件作为资源并且要导入它的字符串名字。

注意：

CWaveFile 样本类会认为它们是“WAVE”或者“WAV”类型的，并会在可执行文件中寻找它们，而不是在动态链接库（DLL）中寻找。

DirectSound 不包含加载声音数据的方法。但是，在 Dsutil.cpp 中使用了许多 SDK 示例程序，实现了几个缓冲区的类，这些类分别从文件、资源和内存中来创建。用 CWaveFile 来创建和加载缓冲区并初始化 DirectSound 有以下几个步骤。

STEP 01 创建一个 CSoundManager 样本类对象。

STEP 02 调用 CSoundManager::Initialize 来创建一个设备对象。

STEP 03 调用 CSoundManager::Create 并把文件或者资源的名字作为参数，或者调用 CSoundManager::CreateFromMemory 函数并把内存地址作为参数。这些方法返回一个 CSound Sample 对象，此对象正好容纳那些数据的一个或几个静态缓冲区。此外，也可以把文件或资源的名字作为参数调用 CSoundManager::CreateStreaming，此方法返回一个代表流缓冲区的类对象。

对上面获取的对象调用 FillBufferWithSound 方法。此方法可以从文件、资源或者是内存来读取数据到缓冲区。而对于流缓冲区，它只读取可以容纳的部分数据。当缓冲区播放数据时，要调用 CStreamingSound::HandleWaveStreamNotification 来更新数据。

3）计算 WAV 声音的播放时间

波形格式的播放时间的长短由数据大小和格式决定，可以通过 CWaveFile::GetSize 和 CWaveFile::GetFormat 方法来获取数据的大小和格式。下面的代码演示了如何计算 WAV 的播放时间，以微秒为单位。

```
DWORD GetSoundLength(LPSTR strFileName)
{
CWaveFile* pWav;
DWORD dwLen = 0;
DWORD dwSize;
WAVEFORMATEX* wfx;
pWav = new CWaveFile();
if (SUCCEEDED(pWav->Open(strFileName, NULL, WAVEFILE_READ))) //打开文件
    {
wfx = pWav->GetFormat();                //得到声音文件格式
dwSize = pWav->GetSize();               //获得文件大小
dwLen = (DWORD) (1000 * dwSize / wfx->nAvgBytesPerSec); //计算播放的时间
pWav->Close();                          //关闭文件大小
}
if (pWav) delete pWav;
return dwLen;
}
```

GetSoundLength()函数的主要功能就是计算WAV的播放时间。首先得到声音文件的地址，并获取声音文件的格式和大小，计算出文件要播放的时间。

3.2.4 使用音效

DirectSound通过DirectX媒体对象（DMOs）提供了声音的音效处理支持。每个DirectX应用程序都可以使用标准的音效，不是标准的DMOs要在系统中注册后才能使用。

所有标准的DMOs都能以DirectSound所支持的任意采样率来处理一个或两个声道的8位或16位PCM数据，交混回响声音不支持8位采样。

为了能够实现音效，一个DirectSound应用程序必须首先调用CoInitialize来初始化COM。在操作之前，要用DirectSoundCreate8来创建设备对象。

用DirectSound API不能创建链式音效，不能将有音效的声音作为输出和另一个包含音效的声音进行混音。下面讲述如何设置音效和使用音效。

1. 在缓冲区中设置音效

用DSBUFFERDESC结构中的DSBCAPS_CTRLFX标志位创建的次缓冲区可以设置任何数据的音效。在设置音效时缓冲区必须停止并且不能被锁定。在DirectX 9.0中，音效不能硬件加速。它可能在一个硬件缓冲区中设置音效，但是效果不明显。

一个非常小的缓冲区音效可能运行不流畅，DirectSound不允许对包含小于150毫秒的数据的缓冲区创建音效。这个值被定义为 DSBSIZE_FX_MIN。下面的例子是在一个缓冲区设置回声音效。

```
HRESULT SetEcho(LPDIRECTSOUNDBUFFER8 pDSBuffer) //pDSBuffer缓冲区的指针
{
HRESULT hr;
```

```
    DWORD dwResults[1];  //每个音效一个元素
     //音效描述
    DSEFFECTDESC dsEffect;
    memset(&dsEffect, 0, sizeof(DSEFFECTDESC));
    dsEffect.dwSize = sizeof(DSEFFECTDESC); //获得DSEFFECTDESC结构的大小
    dsEffect.dwFlags = 0;
    dsEffect.guidDSFXClass = GUID_DSFX_STANDARD_ECHO;
    //设置音效
  if (SUCCEEDED(hr = pDSBuffer->SetFX(1, &dsEffect, dwResults)))
    {
         switch (dwResults[0])  //选择要输出的字符窜
         {
         case DSFXR_LOCHARDWARE:
                 OutputDebugString("Effect was placed in hardware.");
//输出调试字符串“效果被放在硬件之中”
                 break;
         case DSFXR_LOCSOFTWARE:
                 OutputDebugString("Effect was placed in software.");
//输出调试字符串“效果被放在软件之中”
                 break;
         case DSFXR_UNALLOCATED:
                 OutputDebugString("Effect is not yet allocated to hardware or
software.");
//输出调试字符串“效果不分派到硬件或软件”
                 break;
         }
    }
  return hr;
}
```

通过设置缓冲区指针 pDSBuffer 获取 DSEFFECTDESC 结构体的大小，然后设置回声音效，最后显示结果。

2. 音效参数

要设置或者重新获得音效参数，首先必须获得包含音效的缓冲区的接口。下面各个接口中包含了应用程序访问系统都能使用的标准 DMOs 接口。

```
IDirectSoundFXChorus8
IDirectSoundFXCompressor8
IDirectSoundFXDistortion8
IDirectSoundFXEcho8
IDirectSoundFXFlanger8
IDirectSoundFXGargle8
IDirectSoundFXI3DL2Reverb8
IDirectSoundFXParamEq8
IDirectSoundFXWavesReverb8
```

要获得这些接口中的某一个，可以把对应的 ID 传递给包含音效的缓冲区的 QueryInterface 方法。改变音效参数不能总是立即生效，DirectSound 可以在缓冲区声音播放之前处理从播放指针开始的 100 毫秒的声音数据。

3. 标准音效

下面介绍 DirectX 提供的各种标准音效。

- 齐声：表示一个时延比较短暂的并且和原始声音混合产生的重音效果。这种音效由 IDirectSoundFXChorus8 接口实现，它的参数包含在 DSFXChorus 结构中。
- 压缩：表示的是当一个信号的幅度超过某个值时进行衰减。此音效通过 IDirectSoundFXCompressor8 接口实现，在 DSFXCompressor 结构中包含它的参数。
- 失真：表示通过向信号中加入和声来实现，随着所处层级的增加，波形的波峰部分就会变得很清楚。此音效通过 IDirectSoundFXDistortion8 接口实现。在 DSFXDistortion 结构中包含它的参数。
- 回声：一个回声音效会引起整个声音在一个固定延时后重复。此音效通过 IDirectSoundFXEch8 接口实现。在 DSFXEcho 结构中包含它的参数。
- 环境回响：DirectX 对环境回响提供了和交互式的音频特别兴趣小组出版的 2 级交互音频规范（I3DL2）的支持。DirectX 环境回响在 I3DL2 规范里是一个听者属性的实现，而原属性不支持。

注意：

在 DirectSound 实现中，只有声音信号到达对室内音效才能被听到。为了能听到直接路径，应该在另一个没有环境回响的缓冲区里播放此声音。

回响属性通过 IDirectSoundFXI3DL2Reverb8 接口实现，在 DSFXI3DL2Reverb8 结构中包含环境参数。DirectX 中有许多默认的参数设置来描述音频环境范围内的回响特性。很多应用程序可以使用 IDirectSoundFXI3DL2Reverb8::SetPreset 方法来选择这些环境参数中的某一个。为了设置用户自定义的参数，通过用 IDirectSoundFXI3DL2Reverb8::SetAllParamerers 方法来实现，要获得当前的参数通过 IDirectSoundFXI3DL2Reverb8::GetAllParameters 方法来实现。

- 凸缘：是表示一种回声音效，在它的源信号之间间隔非常短并且会随时间而改变。音效结果有时认为是广音，凸缘最初由抓住磁带轴的边缘来改变播放速度。这个音效用 IDirectSoundFXFlanger8 来实现，在 DSFXFlanger 结构中包含它的参数。

3.3 开发环境配置

在进行 DirectSound 开发之前，一定要设置好开发环境，否则编译时会出现很多提示，这些提示很多都找不到定义。

3.3.1 Visual C++基本开发环境配置

DirectX SDK 只是一个开发包，需要在开发环境中进行开发。这里就以 Visual Studio 2005 的 Visual C++ 2005 为例，说明如何配置开发环境。当然，这需要首先安装 Visual C++ 2005。

为了让 VC 能够调用到 SDK 中的头文件和库文件，用户还要做以下的工作：将 DXSDK\inc 和 DXSDK\lib 目录中的所有文件分别复制到 VC 的 include 和 lib 目录中去，这是最简单也是最有效的方法。

3.3.2 DirectSound应用程序的开发环境配置

DirectSound 的开发环境很好设置，简单地说就是包含一些头文件（*.h）和库文件（*.lib），在这些头文件和库文件中包含了很多在开发应用程序中要调用的函数，只要将所需要的头文件和库文件添加到所建的项目中即可，下面将介绍具体的操作步骤。

接下来就设定 DirectSound 程序连接时所必须用到的函数库和头文件。具体过程就以一个项目为例来说明。

STEP 01 打开 Visual Studio 2005，创建一个名为“DirectSound”的 MFC 应用程序项目。具体过程如图 3-12 所示。

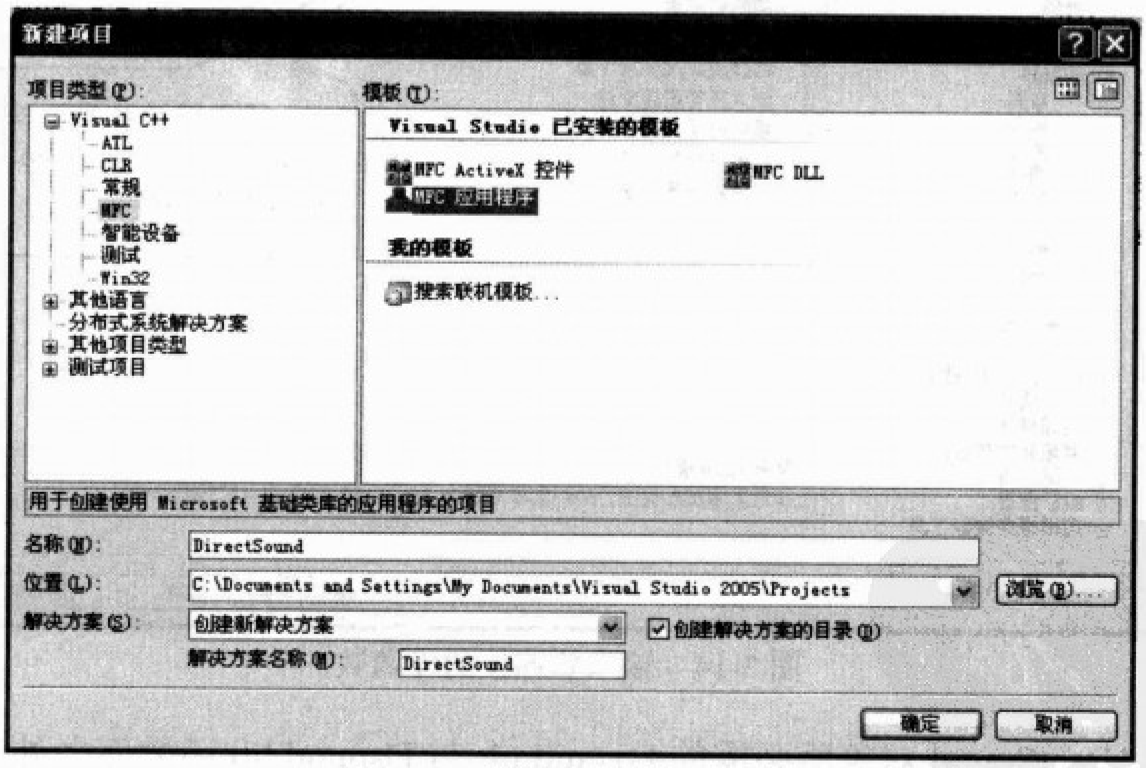


图 3-12 “新建项目”对话框

STEP 02 在生成的界面上的左侧用鼠标右键单击“解决方案资源管理器”中的“DirectSound”选项，选择“属性”命令，如图 3-13 所示。

STEP 03 在弹出的属性页面左侧列表的“链接器”中选择“输入”选项，在右侧的“附加依赖项”中输入建立 DirectSound 对象所需要的 Dxguid.lib、Dsound.lib、Winmm.lib 函数库，如图 3-14 所示。

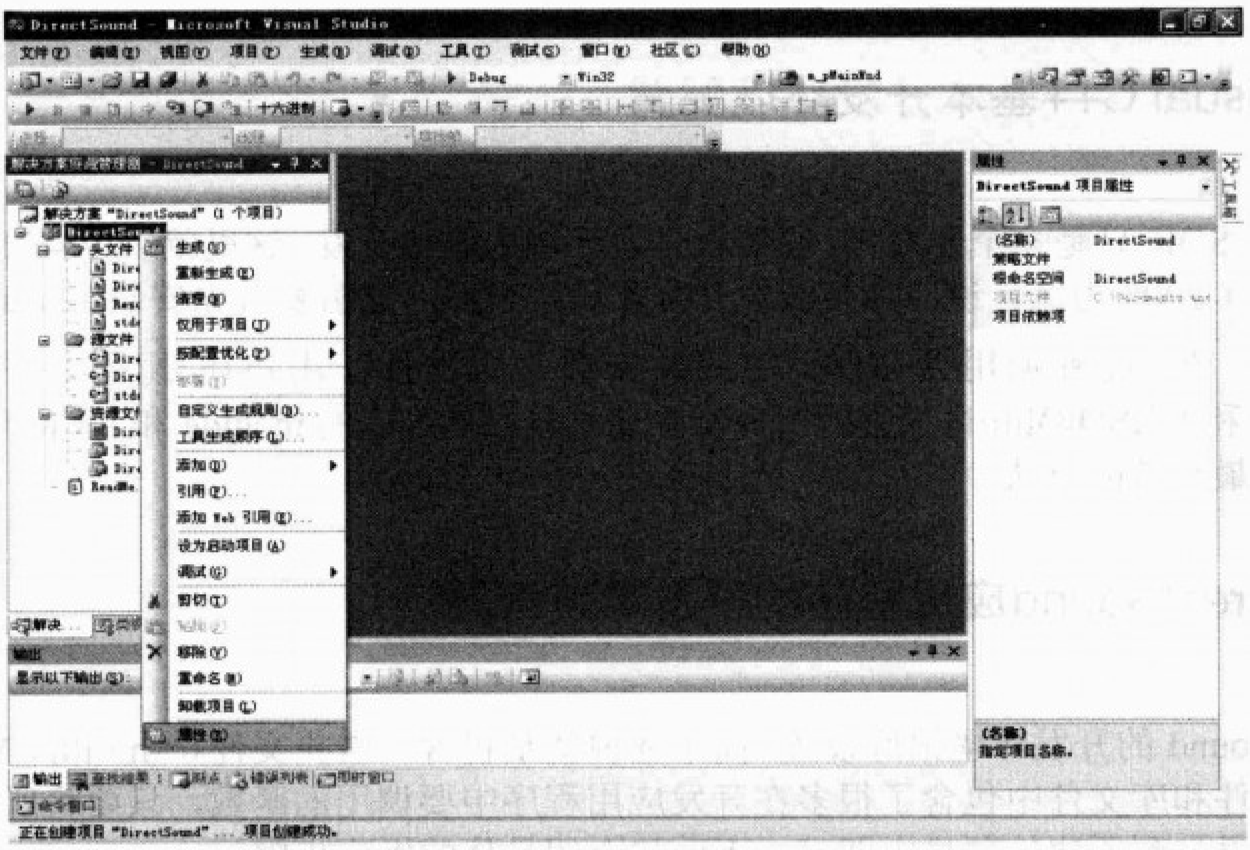


图 3-13　选择“属性”命令

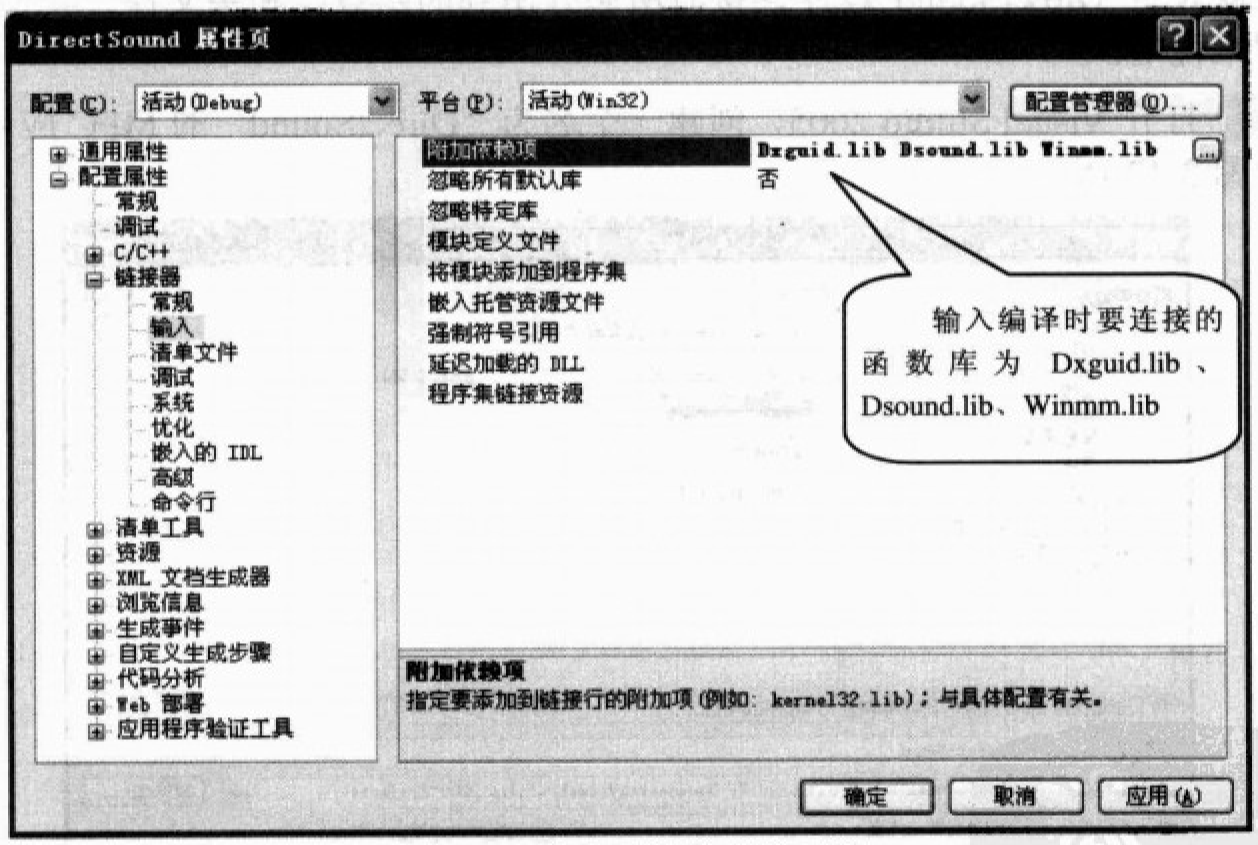


图 3-14　输入要连接的库函数

除了建立 DirectSound 对象所必需的 Dxguid.lib 与 Dsound.lib 函数库之外，Winmm.lib 是 Windows 提供的函数库，它用来处理多媒体文件的函数库。由于在加载 WAVE 文件时会用到，所以在这里必须加入项目的连接中。接下来请打开项目中的 stdafx.h，在其中加入所需要的头文件，如图 3-15 所示。

在这里必须引用 mmsystem.h、mmreg.h、dsound.h 这 3 个头文件，这些头文件是 DirectSound 程序会用到的。在完成了以上这些项目的各项设定之后，接下来就可以开始试着建立与使用 DirectSound 程序了。

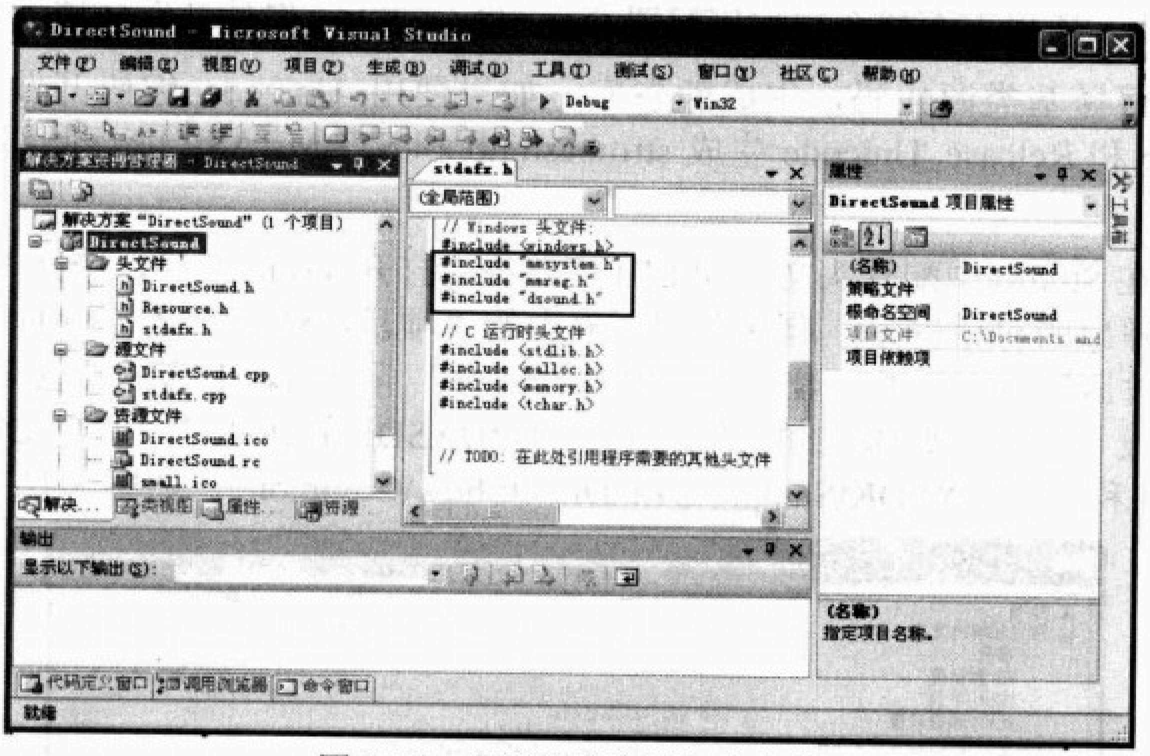


图 3-15 在源程序中添加头文件

仅仅包含 dsound.h 肯定是不够的，一般来说，在用户的工程中包含下面两个文件就够了。

```
#include <mmsystem.h>
#include <dsound.h>
```

如果还想使用 Directsound 的 API 的话，那么就要在 VC 开发环境中添加 dsound.lib 库，为了能使用如 DirectSoundCreate8 之类的帮助性函数，在链接库的搜索路径中要加入这个库所在的路径。

如果程序还提示有很多的外部链接找不到，那么这里建议可以将下面的库都添加到工程中：comctl32.lib、dxerr9.lib、winmm.lib、dsound.lib、dxguid.lib、kernel32.lib、user32.lib、gdi32.lib、winspool.lib、comdlg32.lib、advapi32.lib、shell32.lib、ole32.lib、oleaut32.lib、uuid.lib、odbc32.lib、odbccp32.lib，这些库是从 dsound 所提供的例子中得到的，对读者开发应用程序来说已经足够，至此开发环境已配置好。开发环境配置好之后，就可以在工程中任意使用 DirectSound 提供的接口和函数了。

3.3.3 DirectSound Filter开发环境配置

Filter 是参与 DirectShow 管理和处理数据的各个功能模块。各个 Filter 和 Filter Graph 按照一定的顺序连接成一条"流水线"协同工作。

Filter 是 COM 组件，所以在使用前一定要注册。Filter 的注册程序为 regsvr32.exe。如果带上命令行参数/u，则表示注销；如果带上的是/s，则表示不弹出任何注册/注销成功与否的提示对话框。如果不想让 GraphEdit 找到 Filter，也可以不注册 Filter 信息，且不会影响 Filter 的功能。

这里要特别说一下微软公司所提供的一个 Filter 测试工具——GraphEdit，它的路径是 DXSDK\bin\DXUtils\GraphEdit.exe。通过这个工具，可以很方便地看到 Filter Graph 的运行及处理流程，方便进行程序调试。

需要介绍一下 DirectSound Filter 开发环境的基本配置，因为 Filter 是 DirectShow 中处理

数据的功能模块，所以下面就介绍如何配置 DirectShow Filter 组件开发环境。

STEP 01 编译基类源代码，生成静态库，如果使用 Unicode 编码开发，则分别使用 Debug_Unicode 和 Release_Unicode 生成 strmbasd.lib 和 STRMBASE.lib。此部分在第 7 章将具体讲解。

STEP 02 在 Studio 中配置头文件和 LIB 库路径。打开 Visual C++，选择“工具”→“选项”命令，在弹出的对话框中，选择左侧列表中的“项目和解决方案”中的“VC++目录”子项，然后在对话框右上侧的“显示以下内容的目录”下拉列表中选择“包含文件”选项进行开发环境设置。主要添加的路径有“D:\DXSDK\Include”、“D:\DXSDK\Sample\C++\Common”和“D:\DXSDK\Sample\C++\DirectShow\Common”，如图 3-16 所示。

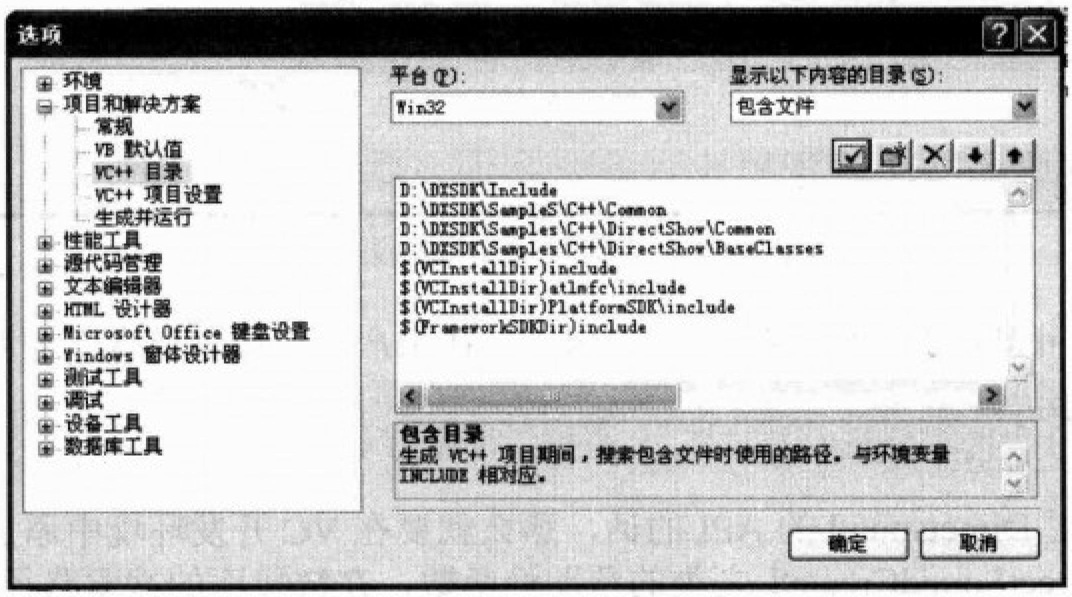


图 3-16　配置包含文件

Lib 库文件路径：在“选项”对话框右上侧的“显示以下内容的目录”下拉列表中选择“库文件”进行配置。主要添加的库文件有“D:\DXSDK\Lib”和“D:\DXSDK\Samples\C++\DirectShow\BaseClasses\Debug_Unicode”，如图 3-17 所示即为所加的库文件。

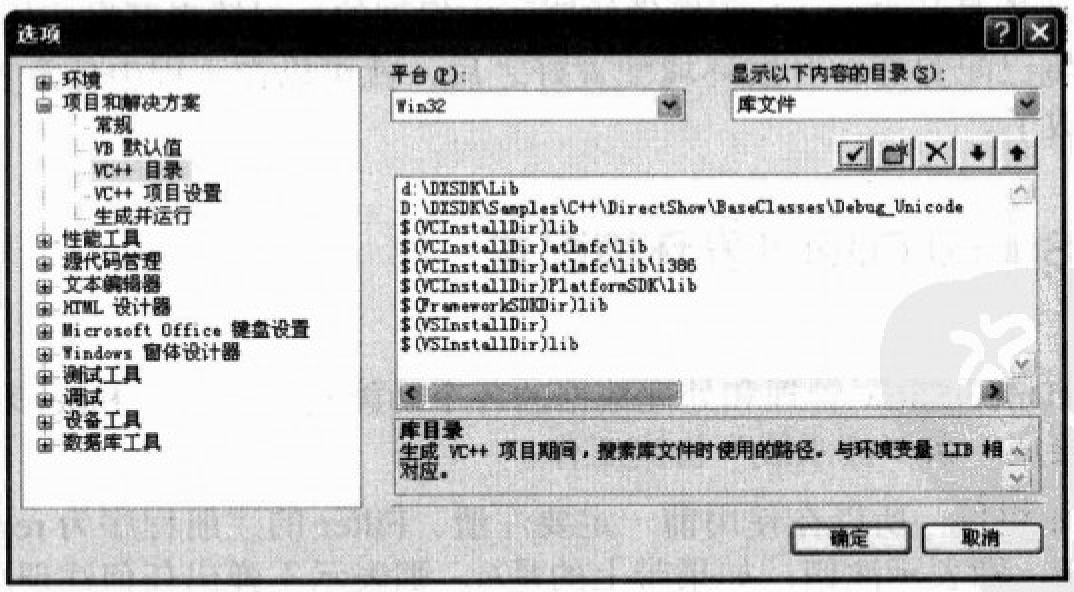


图 3-17　配置库文件

STEP 03 用 Visual C++向导生成 Win32 动态链接库（DLL）作为当前项目。

STEP 04 在 Filter 头文件中包含头文件 streams.h 和 initguid.h。

STEP 05 使用__stdcall 函数调用协议，选择“项目属性”→“C/C++”→“高级”选项，将“调用约定”改为“__stdcall(/Gz)”，如图 3-18 所示。

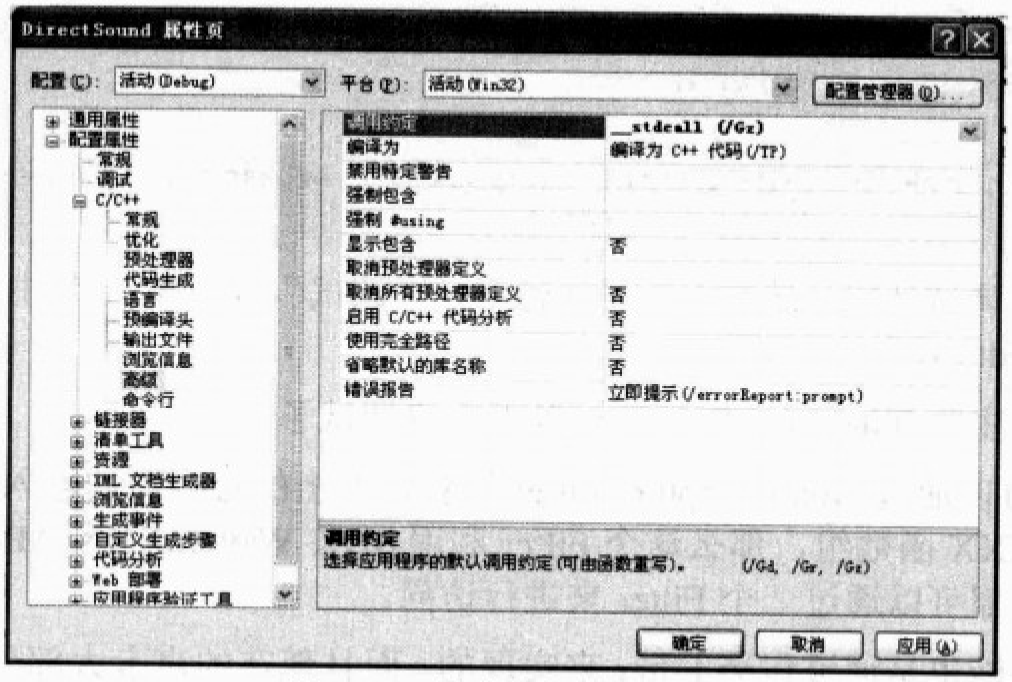


图 3-18 修改“调用约定”

STEP 06 配置 Lib，选择工程属性“连接器”→“输入”选项，在“附加依赖项”中添加 msvcrtd.lib quartz.lib winmm.lib，DEBUG 版本添加 strmbasd.lib，RELEASE 版本添加 strmbase.lib。这几个库文件将在第 7 章详细介绍，配置过程如图 3-19 所示。

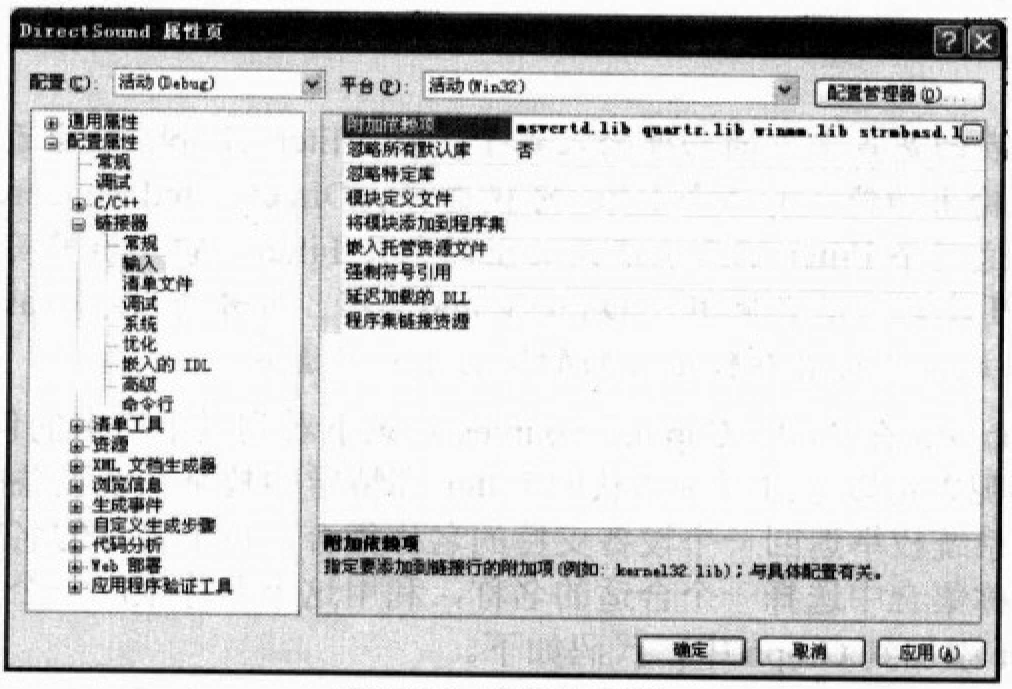


图 3-19 附加依赖项

STEP 07 定义一个 def 文件用于导出 DLL 函数，例如 Filter.def，内容如下。

```
LIBRARY        "Filter.ax"
EXPORTS
    DllMain                    PRIVATE
    DllGetClassObject          PRIVATE
    DllCanUnloadNow            PRIVATE
    DllRegisterServer          PRIVATE
    DllUnregisterServer        PRIVATE
```

STEP 08 定义 DLL 入口函数：

```
extern "C" BOOL WINAPI DllEntryPoint(HINSTANCE, ULONG, LPVOID);
BOOL APIENTRY DllMain(HANDLE hModule,
```

```
            DWORD dwReason,
            LPVOID lpReserved)
{
  return DllEntryPoint((HINSTANCE)(hModule), dwReason, lpReserved);
}
```

STEP 09 配置调试环境，选择“项目属性”→“调试”，在“命令”中输入“D:\DXSDK\Bin\DXUtils\graphedt.exe”。

这样配置后，注册 Filter.ax，然后就可以下断点调试了。

DirectSound 可以通过 Audio Capture Filter 捕获声卡上的音频流的输入。如果设备的驱动程序支持 WaveInXXX 函数组，那么这个 Filter 利用 SDK WaveInXXX APIs 来控制该设备。系统中的每个声卡都可以通过一个 Filter 来进行访问。

声卡的采集功能也是通过包装 Filter 来实现的，而且现在的声卡大部分都有混音的功能。这个 Filter 一般有几个 Input pin，每个 pin 都代表声卡的一个输入设备，如麦克风、CD、MIDI 等。这个输入 Pin 中没有数据的流动，并且它们也不和其他的 Filter 连接。应有程序可以通过它们控制输入。只要驱动程序允许，应用程序就可以通过输入 Pin 来控制输入。如果想要做得更好一些，用户可能要声卡制造商的文档和帮助。

注意：

这些 pin 代表的是声卡上的物理输入端子，在 Filter Graph 中远远不会连接到其他 Filter 上。声卡的输出功能，有两个 Filter 可供选择：DirectSound Renderer Filter 和 Audio Renderer Filter。这两个 Filter 不是上述意义上的包装 Filter，它们能够同硬件交互，是因为它们使用了 API 函数，前者使用了 DirectSound API，后者使用了 WaveOut API。这两个 Filter 的区别，还在于后者在输出音频的同时不支持混音。

可以通过 GraphEdit 在 Audio Capture Sources 目录下看到这个声卡的包装 Filter。如何建立音频的 Graph 图呢？可以通过音频捕获的 Filter 和特定的枚举系统设备的方法来枚举所有的音频捕获设备。系统枚举返回一个设备支持的名称集合，每个名称的名字和设备的名字相关。在后返回的名称集合中选择一个合适的名称，利用这个名称生产一个音频捕获 Filter 的实例，然后将 Filter 添加到 Graph 中，代码如下。

```
IBaseFilter *pSrc = NULL, *pWaveDest = NULL, *pWriter = NULL;
IFileSinkFilter *pSink= NULL;
IGraphBuilder *pGraph;
//创建按Filter图形管理器
CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC_SERVER,
IID_IGraphBuilder, (void**)&pGraph);
//不显示：用系统设备入口创建音频捕获Filter，增加音频捕获Filter到Filter Graph
pGraph->AddFilter(pSrc, L"Capture");
//增加WavDest到文件写指针
AddFilterByCLSID(pGraph, CLSID_WavDest, L"WavDest", &pWavDest);
AddFilterByCLSID(pGraph, CLSID_FileWriter, L"File Writer", &pWriter);
//设置文件名
pWriter->QueryInterface(IID_IFileSinkFilter, (void**)&pSink);
pSink->SetFileName(L"C:\\MyWavFile.wav", NULL);
```

```
//链接Filters
ConnectFilters(pGraph, pSrc, pWavDest);
ConnectFilters(pGraph, pWavDest, pWriter);
```

3.4 调试音频处理程序

当编写好音频应用程序后，就要开始在 Visual C++环境中调试程序了。下面分别介绍一下 Visual C++中调试器和 DirectSound 中的调试技术。

3.4.1 Visual C++调试器

调试程序是程序设计中的一个很重要的环节，往往一个应用程序要经过多次调试后才能保证其正确性。所以可以这么说，程序不是编出来的，而是调试出来的。程序调试一般分为修改源程序中的语法错误和修改程序设计中的错误两个阶段。

Visual C++开发环境提供了一个很重要的集成调试器——Debug 调试器。利用它可以在开发程序的同时方便、动态地进行程序调试，比如可以使程序执行到断点处停止下来，可以通过步进跟踪执行，可以观察变量、表达式和函数调用的关系，可以分析程序的真正的执行情况。如果应用程序没有错误，则开发人员也可以使用调试器来分析程序的执行过程，这对于学习 MFC 应用程序的框架非常有用。

1．启动 Debug 调试器

在启动 Debug 调试器前，首先编译、链接生成一个 Debug 版本的程序。开发环境默认的设置是 Debug 版本，但要生成一个 Release 版本程序后，重新设置为 Debug 版本。通过执行“编译”、“设置配置”命令可以设置当前的程序版本。编译菜单下有一个“启动调试”命令，其中含有启动调试器的命令。如果单击工具栏中的箭头按钮，则程序就会在调试器中运行，到断点处停止。启动调试器后，“调试”菜单取代“编译”菜单出现在菜单栏中，调试出现一个 Debug 工具栏和一些调试窗口。当光标放在程序中的某个函数上后，它的函数原型就会显示出来。

编程时可以生成 Debug 和 Release 两种版本的程序。Debug 版本生成的中间文件及可执行文件放在项目下的 Debug 目录中，目标文件包含所有的调试信息，但程序没有进行优化。Release 版本生成的文件放在 Release 目录中，目标文件不包含任何调试信息，也不能进行程序调试，但程序进行了优化。在编程时一般先生成一个 Debug 调试版程序，经过调试无误后，再编译、链接一个 Release 版的程序。

在调试过程中，屏幕上会同时出现调试器和程序运行窗口。调试时有时需要切换到程序运行窗口，可以通过反复按“Alt+Tab”快捷键进行切换。

2．辅助调试窗口

除了 Debug 工具栏外，Debug 调试器还提供了一些辅助窗口即调试窗口，用于显示程序

的调试信息。调试窗口有许多信息，但通常不需要观察所有的信息。一般情况下，当进入 调试程序时，会打开常见的输出窗口。输出窗口用于显示有关编译和调试的信息，包括编译链接错误提示信息和调试时一些调试宏的输出信息。

变量窗口用于观察和修改某个作用域内所有变量的当前值，调试器可以根据当前程序在运行过程中变量的变化情况自动选择对应的变量。

即使源程序没有语法错误，最后生成的可执行程序也不能正常运行或没有达到设计要求，这类程序设计上的错误称为逻辑设计错误或者缺陷。跟踪调试程序是查找此类逻辑设计错误最常采用的动态方法。跟踪调试的基本原理就是让程序按照源代码的设计流程一步一步地执行，通过观察和分析程序执行过程中数据和代码执行流程的变化来查找程序的逻辑错误。

采用传统的程序设计工具调试程序时，开发人员为了进行跟踪调试，一般需要在程序中人为设置断点，如加入输出变量值的语句。可以利用 Visual C++集成调试器直接进行程序的跟踪调试，而 Debug 调试器是其中最常用的调试器。使用 Debug 调试程序最有效的手段主要有设置断点、逐语句、逐过程、观察数据变化。

有时尽管程序顺利通过了编译链接，但是执行程序时发生错误，或者是程序结果与期望的不同，这时就要借助调试工具以确定发生错误的位置，从而做出适当的处理。

3. 加入断点

断点是进行调试的基本工具。所谓断点，就是程序源代码中的某个位置，在调试器中运行程序时可以强制程序执行到该位置暂时停止运行。

在程序中设置断点的方法很多，只需要在源代码编辑窗口将光标移到要设置断点的代码行，然后在该行的左页边用鼠标单击（或者按下“F9”快捷键）即可。此时在该行的左页边出现一个红色的圆点，如图 3-20 所示。再按“F9”键，红色圆点即被取消。

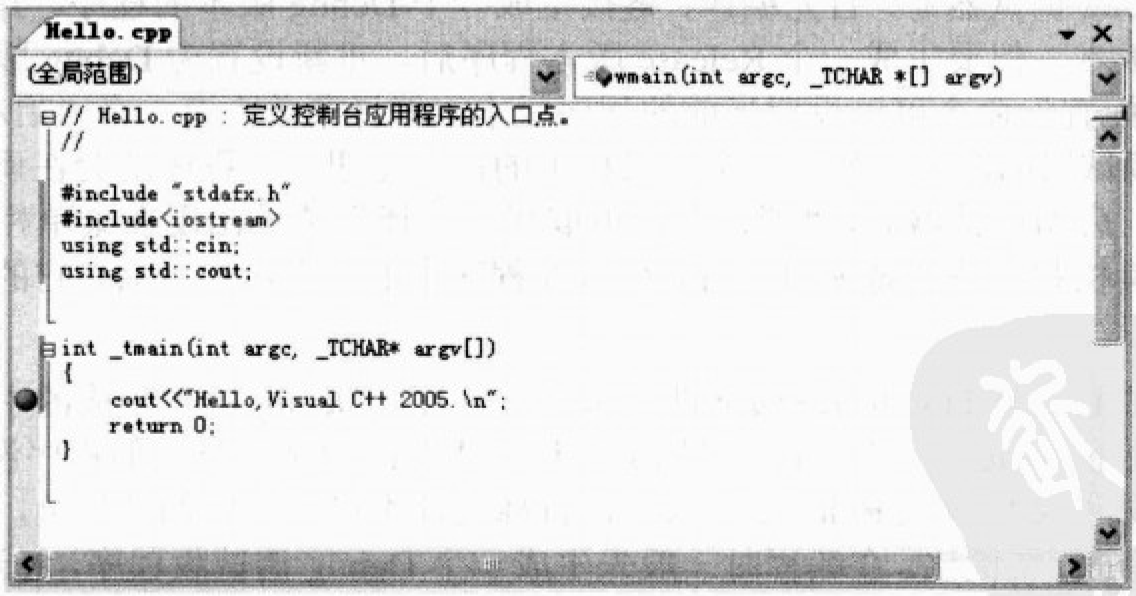


图 3-20 在程序中设置断点

还可以选择“调试”菜单下的“新建断点”命令，弹出“新建断点”对话框以设置新断点，但此种方法比较麻烦，没有前种方法方便。

有时在调试时设置了很多断点，如果一一取消是很麻烦的一件事，这时可以选择“调试”菜单下的“删除所有断点”命令，一次清除所有断点。

当程序运行到断点后程序暂时停止运行，此时可查看或设置程序中的数据内容，并可对代码做小的修改。

4. 步进执行

步进执行是一个最有效的调试手段，它使程序按照源代码的编写流程一行一行执行。步进执行程序的一个主要目的就是观察程序当前的数据变化情况。在 Visual C++调试环境中，步进执行程序时可以利用的信息非常丰富，程序开发人员可以通过观察变量、表达式、调试输出信息、函数等的内容来了解程序的运行程序。

当设置了断点后，选择“调试”菜单下的“启动调试”命令（或者按“F5”键），启动调试程序，程序执行到断点后自动停下来，同时程序的左页边出现一个黄色的箭头，指示程序下一步要执行的程序行，如图 3-21 所示。

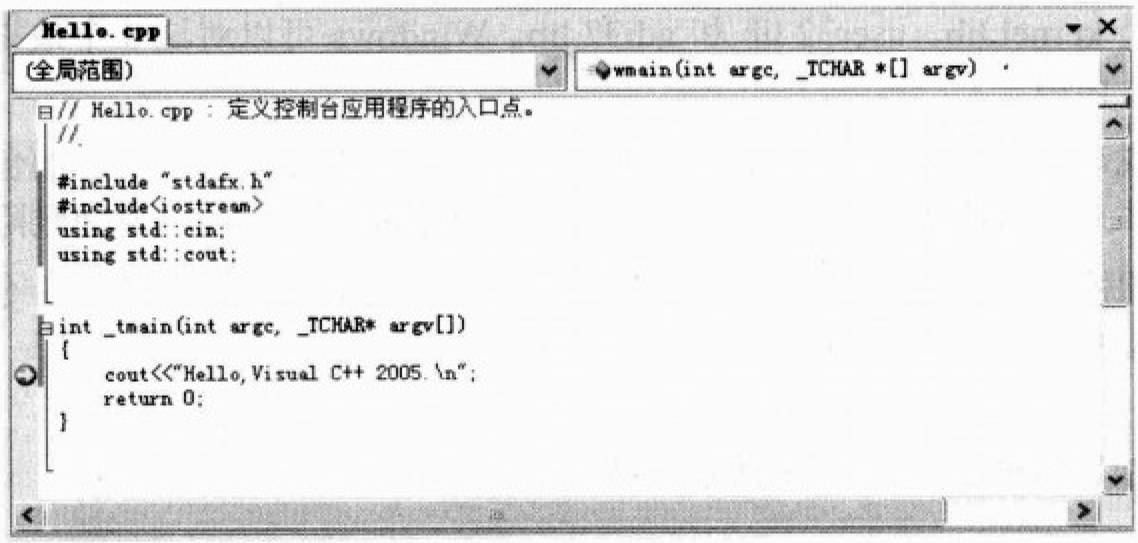


图 3-21　调试状态下的代码编辑框

此时有两种常见的方法实现这些程序。

（1）按“F11”键。这种方式称为逐语句执行，当要执行的语句是简单语句时，它会执行完该语句，然后跳到下一行并停下来等待指示，当要执行的是函数调用时，它会进入该函数，转到调用函数的第一行，并停下来等待指示。比如在图 3-21 中，如果按“F11”键，因为 cout 其实是一个库函数，所以此时程序会打开该库函数所在文件，黄色箭头停留在该库函数源码的第一行，等待指示。若要跳出已进入的函数，则只需按“Shift+F11”快捷键，程序就会自动执行完当前函数剩下的语句，跳回到上一层函数中去。

（2）按“F10”键。这种方式称为逐过程执行。此时当要执行的语句是简单语句时，情况跟按“F11”键一样，如果要执行的语句是函数调用，则它不会进入该函数，而是一直不停地执行完该函数调用，然后跳到下一条语句执行。

可见，要想监视调用函数的运行过程，可按“F11”键来步进执行。如果对函数的执行过程不感兴趣，只是需要获得其返回值，则可按“F10”键来步进执行。

5. 查看编译和调试结果

当程序编写完成并且保存后，就可以编译和调试程序了，首先选择“生成”→“编译”命令（或者按“Ctrl+F7”快捷键），编译解决方案并生成可执行文件。当输出窗口中有错误信息时，在错误信息上双击，即可定位到程序的错误所在行，并再次进行修改。

如果输出窗口中如图 3-22 所示出现：

========== 生成：1 已成功，0 已失败，0 最新，0 已跳过==========

则表示编译顺利完成。

图 3-22 “输出”窗口

其实生成是编译和链接这两个阶段的结合。在正常情况下，在编译阶段，编译器产生一个 obj 文件；而在链接阶段，链接程序会根据 obj 和 lib 文件建立 exe 文件。再选择“项目”→“属性”命令，在弹出的对话框中选择“链接器”选项，可以查看这些 lib 文件列表，在其中最重要的是 kernel.lib、user32.lib 和 gdi32.lib。Windows 可以对这 3 个最重要的动态链接库中的函数进行调用。

在成功生成该项目后，会在项目的目录中生成一个 Debug 文件夹，里面包含了生成的执行文件。此文件夹在每次生成之后会自动更新或重新生成，所以可以随便删除。

如果用户想停止程序的调试，则可以选择“生成”菜单中的“取消”命令，这样就可以退出调试，重新回到程序的编写状态。

3.4.2 DirectSound中常用的调试技术

DirectSound 应用程序开发完成后，就要对应用程序进行编译和调试了，只有熟练了解和运用编译、调试的步骤及技巧，才能缩短 DirectSound 的开发周期。下面主要讲解 DirectSound 中常用的调试技术，主要有以下几点。

1. 编译和链接 DirectSound 工程

在创建的 DirectSound 项目中必须包含 Dsound.h 头文件。如果在项目中同时用到了 DirectSound 和 DirectMusic，那么只需包含 Dmusic.h 就可以了，因为在 Dmusic.h 中包含了 Dsound.h。在 Dound.lib 中，可以使用如 DirectSoundCreate8 之类的帮助性函数，以便直接使用 DirectSound API。

2. 调试 DirectSound 工程

DirectX SDK 中的 DirectSound 动态链接库（DLL）会在程序运行时在调试输出窗口输出调试信息。

可以通过以下方法设置调试级别：打开“控制面板”窗口中的“DirectX”文件，如图 3-23 所示。选择 DirectX 的“属性”页，然后拖动“Debug Output Level”滑块。一共可以设置 0～5 这 6 个调试级别。设置 0 级不输入任何调试消息，而设置为 5 级输出的信息最详细。但在多数情况下设置 2 级或者 2 级以下就可以了，级别越高，相应的错误或警告不会减少，而只会增加一些调试信息，DirectX 属性如图 3-24 所示。

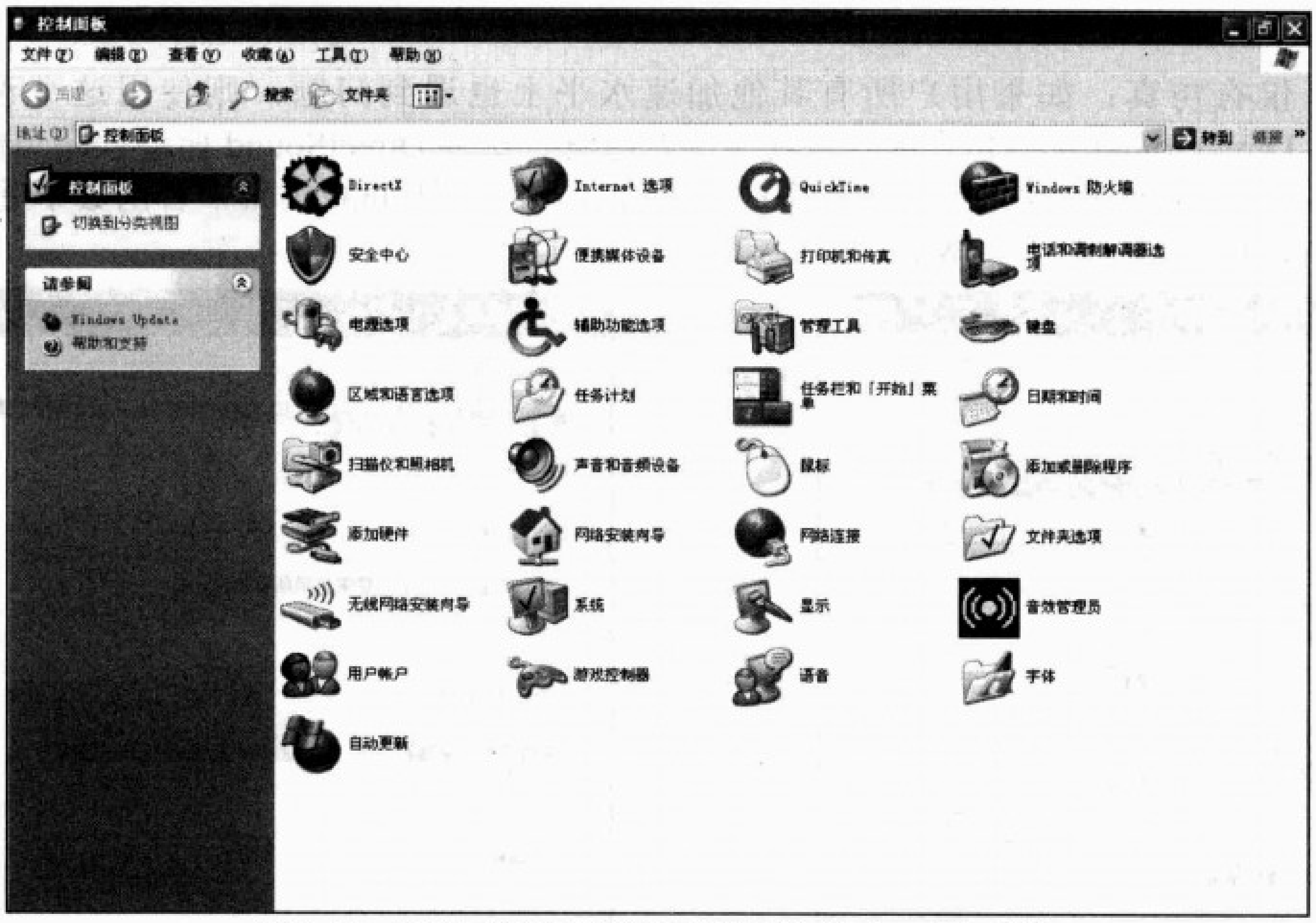


图 3-23 打开“控制面板”面板中的“DirectX”文件

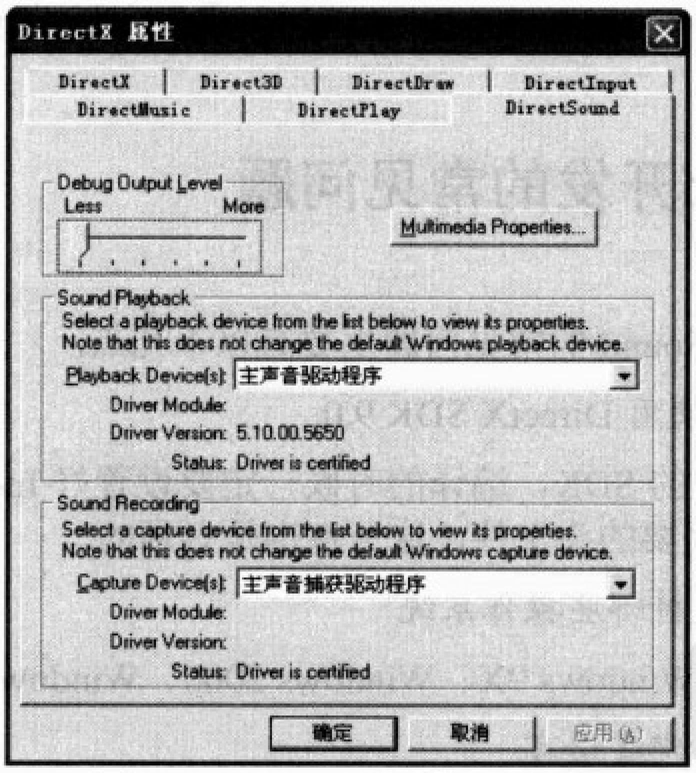


图 3-24 “DirectX 属性”对话框

为了测试，就要调整硬件加速程度，在“控制面板”窗口中“声音和音频设备”文件可以调整系统的 DirectSound 性能。选择“音频”选项卡，然后单击“声音播放”里的“高级”按钮，如图 3-25 所示。出现“高级音频属性”对话框，在“性能”选项卡上，可以拖动“硬件加速”的滑块，如图 3-26 所示。这个滑块可以设定音频硬件加速程度的级别。

- 完全加速：这个选项用来启动所有硬件加速。
- 标准加速：用这个设置只启用标准的 DirectSound 加速功能。在硬件加速中，DirectSound 次缓冲区起作用。

- 基本加速：该设置只启用所需的加速功能。
- 仅有仿真：如果用户所有其他加速水平上也遇到问题，则使用这个设置。让DirectSound进入仿真模式，使系统好像没有安装DirectSound驱动程序一样。所有声音的合成都在用户模式下完成，然后把结果送到Win32波形声音函数中进行处理，这样会使延时大大增加。

图 3-25 “声音和音频设备属性”对话框

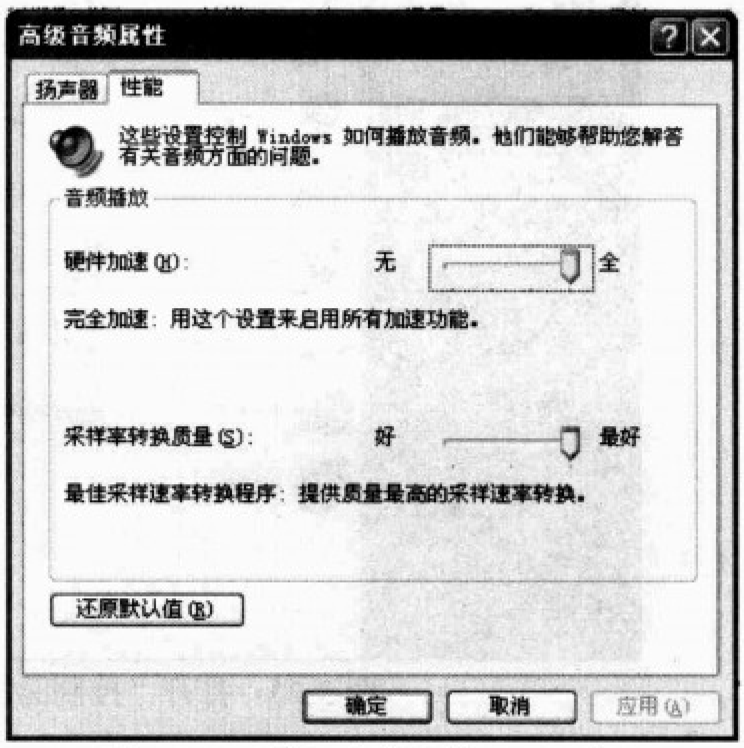


图 3-26 “高级音频属性”对话框

3.5 DirectSound开发的常见问题

Visual C++开发DirectSound的常见问题主要有以下几点。

1）怎样在Visual C++使用DirectX SDK 9.0

首先需要安装DirectX的SDK，编译的时候一定要设置好Include和Link的路径。然后在工程的头文件前添加所需要的头文件，这一点也是必须的。

2）DirectSound通常支持哪些操作系统

DirectSound通常支持Windows 9X、Windows 2000、Windows XP及Vista等操作系统。

3）DirectSound都支持哪些语言

对于DirectSound应用程序的开发，首选的语言是C/C++。同时也支持C#语言和Visual Basic。也可以用Delphi进行编写，但SDK帮助文档没有此方面的内容。

4）如何获取DirectSound Renderer Filter上面的未连接的Pin

文档Audio Renderer（WaveOut）Filter是不支持多路播放的，而DirectSound Renderer Filter则可以。DirectSound Renderer Filter只有1个input pin。如果有多路数据，则要创建多个DirectSound Renderer Filter，声卡会对多路数据自动混合播放。

5）DirectSound循环播放缓冲区时怎么写数据才合适

把DirectSoundBuffer设置为一个数据块的N倍，再设置N个事件点，这样在每个数据

块播放完后就很清楚。也可先填充 2 个数据块，再去播放次数据块，播放完第 1 个再将第 3 个位置写入新数据，播放完第 2 个再将第 4 个位置写入新数据，以此类推，只是需要注意判断到尾部的情况。流式缓冲的写指针需要自己维护，这在 MSDN 的技术文章中有讲述。因为 GetCurrentPosition 返回的是 DirectSound 自己认为“安全”的指针。而事件通知点应该在缓冲区播放之前就设置好。

6）在调试 DirectSound 工程时，有时出现播放速度不正常

可能是调试的环境不一样，即电脑的配置可能导致这种现象，比如双 CPU（双核）机器配置，它播放 DirectSound 程序时的速度会快于正常的速度，此时把电脑的硬件加速功能改成基本加速即可。而在单 CPU 的机器上运行调试 DirectSound 则没有任何问题，当然此时可以把硬件加速调至任何加速级别。

7）怎样可以让 DirectSound 播放出来的声音在音频缓冲更新数据时不出现停顿

一种方法是使用双缓冲区，这样在播放一个缓冲区时填充另一个缓冲区，播放完了再进行切换，DirectSound 支持这种功能。

如果不用缓冲，则系统在两次播放中要处理的事情会很多，而且这些事情是异步触发的，但又有时间顺序的要求。为了最终的同步，系统就要协调处理，这样的开销比较大。如果用了缓冲区，则系统可以在比较空闲的时候把缓冲填好，这样在两次播放切换时系统只需要移动播放的指针就可以了，而且 DirectSound 中一般是将一个连续的缓冲区分成两个部分，在后面的部分播放完成时可以自动回卷（就是跳到起始指针），即真正的指针跳动只有一次。

另一种方法就是使用一个环形的缓冲区，即实现循环缓冲，这样可以边采集边播放。在填充数据的时候使用大缓冲，N 倍于用户的一个播放数据块，用户在播放数据块 1 的时候填充数据块 2，播放数据块 2 的时候填充数据块 1（假设为 2 个数据块）。这只是一个复制数据的过程，不需要其他操作，因此不会有停顿的现象出现。

3.6 本章小结

本章讲解了 DirectSound 的基本概念，介绍了用 DirectSound 进行编程时所需要采取的一般步骤，详细地说明了 DirectSound 的使用原理和方法。Visual C++是 DirectSound 的集成开发环境，介绍了 Visual C++和 DirectSound 开发环境的基本配置。讲解了 Visual C++和 DirectSound 对音频的调试技术。在用 Visual C++开发 DirectSound 程序中会遇到不少问题，本章最后列举了用 Visual C++进行 DirectSound 开发常遇到的问题及解决的办法。

第 4 章

打造自己的音频播放器

DirectSound 可以使应用程序用很少的延时来播放声音文件，同时也可以使程序对声音硬件资源进行控制。DirectSound 可以实现播放 WAV 文件格式的声音文件，还具有同时播放多路声音（8 个及 8 个以上）的强大功能。正是由于 DirectSound 在音频播放功能上比 MCI、PlaySound()函数更具有优越性，所以用 DirectSound 开发可以得到意想不到的结果。本章就是利用 DirectSound 来开发属于自己的音频播放器。

本章重点

- 系统设计和分析
- 缓冲区和通知基础
- 播放文件
- 系统效果演示
- DirectSound 开发常见问题

4.1 系统设计和分析

本案例制作了一个自己的音频播放器，通过 DirectX 的 DirectSound 组件中的 API 函数来实现播放、停止、暂停等基本操作，DirectSound 编程技术在多媒体编程中经常会用到。通过本案例的学习，读者可以掌握 DirectSound API 各个接口函数的使用方法，并能自己动手编写一个功能更齐全的音频播放器。

DirectSound 是微软提供的 DirectX SDK 开发包中的一个组件，可以使用其进行声音的多媒体编程。本实例介绍用 DirectSound 来播放 WAV 格式的声音文件。API 提供了 3 种方法来播放 WAV 文件。

1）MCI

MCI（Media Control Interface，媒体控制接口）向 Windows 程序提供了在较高层次上控制媒体设备接口的能力。程序不必关心具体设备就可以对 CD、波形音频设备、视频播放设备和 MIDI 设备等媒体设备进行控制。MCI 功能强大，不仅可以播放 WAV 文件，也可以播放 MIDI 及 CD 音频。

2）PlaySound()函数

PlaySound()函数可以通过单行编码来播放 WAV 格式声音，此函数受到两个限制：必须将声音数据完整地载入物理内存；数据格式必须被所配置的某一音频驱动器支持。因此，PalySound()函数适用于 100KB 以下的文件。

3）低级 WAV 音频设备

这些音频设备可以运行完全控制 WAV 数据的应用文件。前 2 种设备使用方法简单，但是无法实现 2 个或 2 个以上的 WAV 文件的播放，最后一种方法实现起来非常复杂。而用 DirectSound 可以同时播放 8 个及 8 个以上的 WAV 文件，容易使用，可使多媒体更加生动，但是它不支持 WAV 文件的记录及存储，这是唯一的缺点。DirectSound 播放音频的整个流程如图 4-1 所示。

图 4-1　DirectSound 播放音频数据流程

设计原理分析：将次缓冲区的声音数据输出到混声器进行混频处理，同样输出数字化的混频声音数据到主缓冲区。主缓冲区位于声卡的内存上，默认采用双声道、8bit 和 22 050Hz 方法播放声音。主缓冲区中的声音数据转换为模拟信号，送入扬声器中进行播放。这个是声音的播放过程，DirectSound 提供了相应的封装代码。基本编程形式是，取得声卡中的一块内存或者将系统内存作为次缓冲区，装入声音数据到次缓冲区，设置声音的格式，调用次缓冲区的 Play 方法和 Stop 方法，播放或停止播放声音，这一步实际是次缓冲区数据到混声器的传递。然后数据在混声器处理后进入主缓冲区，声卡自动将声音通过扬声器播放出来。

设计思路：先创建一个代表声卡的 DirectSound 对象，然后设定协作级别告诉 Windows 所要使用的权限；声音

在 DirectSound 中是用缓冲区来播放的，在创建的主缓冲区设置播放格式来改变声音播放的品质和效果；声音播放前，数据应存储在创建的次缓冲区等待加载；要将声音加载到次缓冲区中，就要对次缓冲区进行写入操作，必须先锁定缓冲区，待把数据加载完成后解除锁定，然后即可对声音进行播放控制，直到停止播放，退出程序。

4.2 缓冲区和通知基础

在介绍缓冲区的概念之前，先了解一下 DirectSound 在缓冲区中所保留的两个指针，播放指针和写指针。这两个指针分别表示播放位置和写位置相对于缓冲区头的偏移字节数，而不是它们的绝对内存地址。

IDirectSoundBuffer8::Play 方法总是从播放指针的开始位置播放声音数据。当缓冲区被创建时，播放指针被置为 0。当缓冲区中的数据被播放时，指针移动并一直指向下一个要播放的字节。当缓冲区被停止播放时，指针停在当时播放的位置。

写指针指向一个可以安全地把数据写入缓冲区的位置。在播放指针和写指针之间的数据块肯定要播放，并且也不能安全地改变。

可以把缓冲区看做是一个钟表面，写入数据就是按顺时针方向，而播放指针和写指针分别以相同的速度围绕这个时钟表面运动，写指针总是保持在播放指针之前很近的位置。如果播放指针指向 1，写指针指向 2，则只有在 2 之后写入数据才是安全的。在 1 和 2 之间的数据可能已经被 Di0rectSound 排成队列等待播放了，并且不能不能被移动。

写指针随着播放指针而移动，并不把数据写入缓冲区。如果是数据流的形式，那么应用程序就要自己维持自己的指针指向缓冲区，从而来确定下一个将要写入的数据块。

一个应用程序获得播放指针和写指针，通过调用 IDirectSoundBuffer8::GetCurrentPosition 方法来实现，IDirectSoundBuffer8::SetCurrentPosition 方法来设置播放指针，应用程序并不能控制写指针。

为了尽可能准确地保证播放指针，当创建一个次缓冲区时应设置 DSBCAPS_GETCURRENTPOSITION2 标志位来实现。

4.2.1 缓冲区的概念

在 DirectSound 中是使用缓冲区来播放声音的，下面来说明 DirectSound 中所使用的两种缓冲区类型。

1）主缓冲区（Primary Buffer）

主缓冲区可以为一个 DirectSound 用来播放声音、产生混音效果的区域，它有一个默认的播放格式（8bit、22kHz），而声音文件在播放时便会依次输出。

主缓冲区在建立 DirectSound 对象时会自动生成，如 16bit、44kHz 就要自行建立主缓冲区并设定其播放格式，且在前面设定协调级别时，标志必须设定为 DSSCL_PRIORITY。

对于所设定的播放格式，若是用户的声卡不支持，并不会影响程序的执行。DirectSound会自动按照用户计算机的配置来调整最佳的输出效果。这个优点省去了程序开发人员检查每个不同用户、不同计算机硬件配置的麻烦。

（2）次缓冲区（Secondary Buffer）

次缓冲区则用来存储要播放的声音文件，可以建立数个次缓冲区来存放多个要播放的声音文件。

当建立了次缓冲区并在其中加载声音文件后，便可播放其中的内容。播放的方式很简单，只要调用次缓冲区对象的 Play 方法，声音就会自动地被丢到主缓冲区中并进行播放。而若是在同一时间内要播放多个声音文件，如同时播放背景音乐与事件声音，那么播放的方法也是一样的。每个存储声音的次缓冲区调用 Play 方法，声音就会被丢到主缓冲区，接着在主缓冲区中对这些声音进行混音的动作，最后再输出。如此就可在同一时间听到多种不同声音的播放。

对于次缓冲区中所存储的声音不仅可以进行播放，还可以直接调用方法来改变音量、调整声道、停止或循环播放等。

4.2.2 缓冲区的通知

当缓冲区的播放指针到了某些定点或者播放结束时，DirectSound 会通知应用程序。而这个功能通常在两种情况下有用。

- 应用程序必须在对播放进行一些相关动作，如播放结束后，播放另外一个缓冲区。
- 为了把数据以流的形式放入缓冲区，应用程序需要知道播放指针是否到达了一些预先设定的点，以便把新的数据放入缓冲区，来代替旧的数据。

通过调用 IDirectSoundNotify8::SetNotificationPositions 方法可以在缓冲区中设定任何数目的通知点，并对这些点进行相应的操作。在缓冲区播放过程中不能调用上述方法。

首先要设置通知点，就要得到一个 IDirectSoundNotify8 接口的指针，通过缓冲区的对象 QueryInterface 方法来实现。一个独立的 IDirectSoundNotify8 接口指针都是从用户要设置通知点的每个缓冲区获得的，对于每一个通知点位置，用 Win32CreateEvent 函数来创建一个事件对象，然后把这个事件对象的句柄放入 DSBPOSITIONNOTIFY 结构中的 hEventNotify 成员中，在结构的 dwOffset 成员中设定相对于缓冲区头偏移的目标通知点，如果向设置更多的通知点，就把结构体数组的地址传递给 SetNotificationPositions 方法。

下面的示例设置了一个通知点的位置。

```
HRESULT SetStopNotification(HANDLE hMyEvent, //定义设置通知点位置函数的结构
      LPDIRECTSOUNDBUFFER8 lpDsbSecondary)
{
  LPDIRECTSOUNDNOTIFY8 lpDsNotify;
  DSBPOSITIONNOTIFY PositionNotify;
  HRESULT hr;
  if (SUCCEEDED(
      hr = lpDsbSecondary->QueryInterface(IID_IDirectSoundNotify8,
      //获取接口指针
              (LPVOID*)&lpDsNotify)))
```

```
    {
      PositionNotify.dwOffset = DSBPN_OFFSETSTOP;     //通知点的位置偏移量
      PositionNotify.hEventNotify = hMyEvent;         //通知点事件句柄
      hr = lpDsNotify->SetNotificationPositions(1, &PositionNotify);
      //设置通知点位置
      lpDsNotify->Release();
    }
   return hr;
 }
```

当缓冲区停止播放时这个事件会被通知，因为一个非循环播放的缓冲区已经播放到结尾了或者是应用程序调用了 IDirectSoundBuffer8::Stop 方法。假设 hMyEvent 是一个由 CreateEvent 返回的句柄，则 lpDsbSecondary 用 DSBCAPS_CTRLPOSITIONNOTIFY 标志位来创建了一个次缓冲区，本程序则详细地说明了此点。

4.3 播放文件

在播放声音文件之前，必须通过在缓冲区中设置指针来告诉应用程序播放声音的具体位置以便控制，这在上一节中已详细介绍。但是要播放声音文件还有几个阶段，下面则详细介绍了创建播放文件的几个必需步骤和方法。

4.3.1 创建DirectSound对象

播放声音文件首先要创建 DirectSound 对象，目的就是获得次缓冲区，同时也要对设备进行协调级别设置，来规定硬件的使用权限。

1. 创建 IDirectSound 对象

要取得次缓冲区，首先需要创建一个 IDirectSound8 接口对象。为此，调用 DirectX API 函数提供的 DirectSoundCreate8 进行创建。该函数用到 dsound.h 和 dsound.lib 文件，如下是它的使用原型。

```
HRESULT WINAPI DirectSoundCreate8
(
LPCGUID lpcGuidDevice,      //声卡设备全局标志，一半是NULL，表示默认的声卡
LPDIRECTSOUND8 *PPDS8,      //返回的IDirectSound8接口对象指针
LPUNKNOWN pUnkOuter         //必须为NULL
);
```

直观地说，IDirectSound8 对象代表声卡设备。

2. 设置设备协调级别

DirectSound 使用了声卡，当然声卡也是 Windows 管理的资源之一，因此在建立了

DirectSound 对象之后，便要设定协调级别来告诉 Windows 所要使用的硬件权限。IDirectSound 声卡设备创建出来以后，需要调用 DirectX API 函数 SetCooperativeLevel，进一步设置设备的协调级别，以决定声卡设备以哪种方式与 Windows 其他程序分享。

```
HRESULT SetCooperativeLevel
(
HWND hwnd,        //窗口句柄
DWORD dwLevel     //协调级别
);
```

其中，参数 hwnd 为当前应用程序的窗口句柄。dwLevel 参数为协调级别，具有如下几个取值。

- DSSCL_EXCLUSIVE：对于 DirectX 8.0 以上版本，此标志与 DSSCL_PRIORITY 具有相同的作用。对于以前的版本，此标志设置声卡设备为当前程序独占。
- DSSCL_PRIORITY：具有优先设置 DirectSound 设备的权限。例如，可调用 IDirectSoundBuffer8 接口 SetFormat 函数这只主次缓冲区的播放形式。在此标志下，DirectSound 必然是独占的，否则不可能重新设置播放格式，因为其他使用该设备的程序是不知道新格式的。
- DSSCL_NORMAL：正常的协调层级标志，其他程序可共享声卡设备进行播放。在此标志下，主缓冲区的播放格式不允许修改，而且会默认在声卡内存中申请到一个 8bit 和 22kHz 的单声道主缓冲区。
- DSSCL_WRITEPRIMARY：可写主缓冲区，此时次缓冲区就不可能进行播放处理，即不能将次缓冲区的数据送进混声器，再输出到主缓冲区上。这是最完全控制声音播放的方式，意味着声卡的混音器被弃置，需要自行处理声音混合。

下面的代码给出了调用 IDirectSound 接口的 SetCooperativeLevel 函数，此函数的原型定义为：

```
HRESULT SetCooperativeLevel(
   HWND hwnd,          //窗口句柄
   DWORD dwFlags       //标识符
);
```

设置声卡设备协调级别的示例如下。

```
if(FAILED(m_pDirectSound->SetCooperativeLevel(hwnd,DSSCL_NORMAL))
   return false;    //调用协调级别函数SetCooperativeLevel()
```

在协调级别函数 SetCooperativeLevel()函数中，hwnd 是窗口的句柄，设置的协调级别的标志为 DSSCL_NORMAL，即正常的协调级别。

4.3.2 创建播放缓冲区

当创建了 DirectSound 对象和协调级别设置完成后，就要创建声音的播放缓冲区了，首先创建主缓冲区和次缓冲区，然后把声音加载到次缓冲区等待播放。

1. 主缓冲区的创建

了解了 DirectSound 中缓冲区的概念之后，接下来说明建立主缓冲区与设定播放格式的方法。上面已经提出，设备级别设置为 DSSCL_NORMAL，将自动分配一块声卡内存作为主缓冲区，并使用双声道、8bit 和 22kHz 方式进行声音播放。所以，主缓冲区一般是现成的，可略去在声卡内存中再进行主缓冲区的创建分配。

若设置主缓冲区的新的播放格式，则可调用 IDirectSound8 接口的 CreateSoundBuffer 函数，从声卡内存中分配出缓冲区。当然，此时不能再使用 DSSCL_NORMAL 方式的设备协调级别，因为该协调级别不允许进行主缓冲区的格式修改，次级别系统会自动进行分配的。

```
HRESULT CreateSoundBuffer
(
LPCDSBUFFERDESC pcDSBufferDesc,         //缓冲区结构
LPDIRECTSOUNDBUFFER *ppDSBuffer,        //返回的缓冲区指针
LPUNKNOWN pUnkOuter                     //必须为NULL
);
```

注意：

pcDSBufferDesc 参数。它是一个 DSBUFFERDESC 结构体变量指针，该结构体给出了缓冲区的一些描述信息。下面是 DSBUFERDESC 结构体的定义。

```
typedef struct
{
DWORD dwsize;                   //结构体的大小
DWORD dwFlags;                  //缓冲区的用途标志
DWORD dwBufferBytes;            //缓冲区的大小
DWORD dwReserved;               //保留未用，必须为0
LPWAVEFORMATEX lpwfxFormat;     //声音的格式属性
GUID   guid3DAlgorithm;         //3D声效算法
}DSBUFFERDESC, *LPDSBUFFERDESC;
typedef const DSBUFFERDESC *LPCDSBUFFERDESC;
```

注意：

dwSize 参数表示结构体的大小，单位是字节。dwFlags 参数是一个标识符，表示缓冲区的用途，意义如表 4-1 所示。dwBufferBytes 参数为缓冲区的大小，主缓冲区的大小必须设置为 0。dwReserved 参数保留，必须设置为 0。

表 4-1　dwFlags 标识符描述

标识符	说明
DSBCAPS_CTRL3D	缓冲区具有 3D 音效控制能力，不能与 DSBCAPS_CTRLPAN 标志一起使用
DSBCAPS_CTRLFREQUENCY	缓冲区可设置采样频率

（续表）

标识符	说明
DSBCAPS_CTRLFX	缓冲区支持特效（Effects Processing）处理，但缓冲区必须足够大，以致可容纳更细微时间点上的数据
DSBCAPS_CTRLPAN	缓冲区可控制声道
DSBCAPS_CTRLPOSITIONNOTIFY	缓冲区具有播放位置通知能力
DSBCAPS_CTRLVOLUME	缓冲区可设置音量大小
DSBCAPS_GLOBALFOCUS	缓冲区是一个全局声音资源，当前程序切换到其他程序(包括使用 DirectSound)，仍可继续播放声音
DSBCAPS_LOCDEFER	缓冲区可绑定为硬件或软件内存来播放声音
DSBCAPS_LOCHARDWARE	缓冲区必须使用硬件的混声器，如果硬件不支持混音或硬件内存不支持建立缓冲区，则执行 CreateSoundBuffer 函数将会失败
DSBCAPS_LOCSOFTWARE	缓冲区使用软件内存或使用软件混音
DSBCAPS_MUTE3DATMAXDISTANCE	超过最大的可听距离，将停止声音播放
DSBCAPS_PRIMARYBUFFER	说明缓冲区用做主缓冲区，如果此标志没有说明，则用做次缓冲区
DSBCAPS_STATIC	自动使用硬件内存作为缓冲区
DSBCAPS_STICKYFOCUS	当程序切换到其他不使用 DirectSound 的程序时，缓冲区仍播放声音，但无法正常进行其他处理

可以看出，主缓冲区可通过 dwFlags 参数的 DSBCAPS_PRIMARYBUFFER 标志值进行创建。

lpwfxFormat 参数指出声音的格式属性，下面是 WAVEFORMATEX 结构体的定义说明。

```
typedef struct _ WAVEFORMATEX
{
WORD wFormatTag;           //格式类型，表示压缩算法
WORD nChannels;            //声道数（通常是1~2个声道）
DWORD nSamplesPerSec;      //采样频率
DWORD nAvgBytesPerSec;     //每秒采样字节数
WORD nBlockAlign;          //块对齐大小
WORD wBitsPerSamplel       //采样位数
WORD cbSize;               //附加的数据大小
}WAVEFORMATEX;
```

对于主缓冲区来说，lpwfxFormat 参数必须设置为 NULL，这样主缓冲区可用下面的简单代码创建出来。

```
DSBUFFERDESC bufferDesc;
ZeroMenory (&bufferDesc, sizeof (DSBUFFERDESC) );
bufferDesc.dwSize=sizeof(DSBUFFERDESC);
bufferDesc.dwFlags=DSBCAPS_PRIMARYBUFFER;          //表示主缓冲区
bufferDesc.dwBufferBytes=0;                        //主缓冲区必须设置为0
```

```
    bufferDesc.lpwfxFormat=NULL;                 //主缓冲区必须设置为NULL
    //取得主缓冲区
    IDirectSoundBuffer *pDirectSoundBuf=NULL;
    if(FAILED(m_pDirectSound->CreateSoundBuffer(&bufferDesc,&pDirectSoundBuf,NUL
L)))
    return false;
```

主缓冲区的格式修改通过 IDirectSoundBuffer 接口函数 SetFormat 来实现，如下是它的使用原型。

```
    HRESULT SetFormat
    (
    LPCWAVEFORMATEX pcfxFormat  //声音格式属性
    );
```

下面的代码设置了主缓冲区使用单声道、16bit 和 22.05kHz 格式播放声音。

```
    WAVEFORMATEX waveFormat;
    ZeroMemery(&waveFormat,sizeof(WAVEFORMATEX));
    waveFormat.wFormatTag=WAVE_FORMAT_PCM;
    waveFormat.nChannels=1;                  //单声道
    waveFormat.nSamplesPerSec=22050;         //22.05kHz的采样频率
    waveFormat.wBitsPerSample=16;            //16bit
    waveFormat.nBlockAlign=(waveFormat.wBitsPerSample/8) *waveFormat.nChannels;
    // 计算采样快对齐大小
    waveFormat.nAvgBytesPerSec=waveFormat.nSamplesPerSec *waveFormat.nBlockAlign;
                                             //计算平均每秒采样字节数
    if(FAILED(pDirectSoundBuf->SetFormat(&waveFormat)))
                                             //设置缓冲区格式
        return false;
```

2. 创建次缓冲区

建立了主缓冲区之后，接下来就要建立次缓冲区。在 DirectSound 程序中至少要有一个次缓冲区来存储要播放的声音。而次缓冲区的建立必须要参考所加载声音文件的两个信息：声音文件的格式与波形数据的大小。因此，必须根据实际的声音文件大小，创建一个相应大小的次缓冲区。

如图 4-2 所示是.wav 文件的结构说明。.wav 文件依据称为 RIFF 的资源互换文件格式（Resources Interchange File Format）来存放声音数据。文件包括 3 个块（Chuck），分别是 RIFF 块、fmt 块和 data 块。

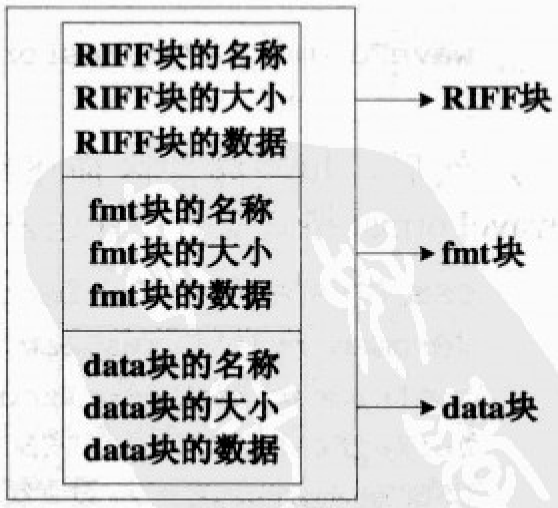


图 4-2 .wav 声音文件的格式

每个块由块名称、块的大小及块的数据构成。比如，一个.wav 文件，RIFF 块的名称为“RIFF”，大小为 4 个字节，RIFF 块的数据为“WAVE”，这个字符串占用 4 个字节；fmt 块的名称为“fmt”，后面有一个空格，为 4 个字符，块的大小为 16 个字节；data 块的名称为“data”，该块的大小由实际的文件中的声音数据大小来决定。

为了读取.wav 文件的声音格式数据，并将实际声音数据装入次缓冲区中，定义如下的一个对应于.wav 文件头的结构体 AUDIO_CONFIG。

```
typedef struct _AUDIO_CONFIG {
  char RiffID[4];                        //'RIFF'块
  char WaveID[4];                        //.wav文件应为'WAVE'类型
  char DataID[4];                        //'data'块
  char FormatID[4];                      //'fmt'块
  WORD WaveformChunkSize;                //4个字节
  WORD FormatChunkSize;                  //16个字节
  WORD wFormatTag;                       //.wav文件应为WAVE_FORMAT_PCM
  WORD nChannels;                        //声道数
  DWORD nSamplePerSec;                   //采样频率
  DWORD nAvgBytesPerSec;                 //每秒采样的字节数
  WORD nBlockAlign;                      //块对齐大小
  WORD wBitsPerSample;                   //bit位数(8bit或bit)
  DWORD DataSize;                        //data块的大小(实际的声音数据)
} AUDIO_CONFIG;
```

先读取文件头信息到 AUDIO_CONFIG 结构体变量，为后面处理做准备。次缓冲区的创建与主缓冲区的创建方法差不多，只是有一些参数设置需要作出修改。

首先，次缓冲区需要根据 WAV 文件头的数据往 WAVEFORMATEX 结构体中填写信息，来反应声音数据的基本信息，如下面代码所示。

```
WAVEFORMATEX waveFormat; //wave声音格式结构体
ZeroMemory(&waveFormat,sizeof(WAVEFORMATEX));
waveFormat.wFormatTag=WAVE_FORMAT_PCM;          //设置声音格式
waveForamt.nChannels=pWaveHeader->Channels;  //声道数
waveForamt.nSamplesPerSec=pWaveHeader->SampleRate;     //采样率
waveFormat.wBitsPerSample=pWaveHeader->BitsPerSample; //位数
waveForamt.nBlockAlign=(waveFormat.wBitsPerSample/8)*waveFormat.nChannels;
                                                        //块对齐大小
waveForamt.nAvgBytesPerSec=waveFormat.nSamplesPerSec*waveFormat.nBlockAlign;
                                                        //平均每秒采样字节数
```

然后，填写缓冲区描述结构体信息，提供音量控制、声音数据大小和格式等，将 waveFormat 格式变量设置进去。

```
DSBUFFERDESC bufferDesc;    //填写缓冲区缓冲区描述结构体
ZeroMemory(&bufferDesc,sizeof(DSBUFFERDESC));
bufferDesc.dwSize=sizeof(DSBUFFERDESC);
bufferDesc.dwFlags=DSBCAPS_CTRLVOLUME|DSBCAPS_GLOBALFOCUS|
DSBCAPS_STATIC; //设置缓冲区音量大小、缓冲区为全局资源
及硬件内存作为缓冲区
bufferDesc.dwBufferBytes=pWaeHeader->DataSize;     //声音文件大小
bufferDesc.lpwfxFormat=&waveFormat;                //声音格式
```

其中结构体 DSBUFFERDESC 中的参数 dwFlags 是创建缓冲区时要用到的标志位，其设定标志和说明在前面已经介绍过，可参照。

调用 IDirectSound8 接口的 CreateSoundBuffer 函数创建一个 IDirectSoundBuffer 对象，利用 IDirectSoundBuffer 接口的 QueryInterface 函数查找返回 IDirectSoundBuffer8 接口对象。

具体代码如下所示。

```
if(FAILED(m_pDirectSound->CreateSoundBuffer(&bufferDesc,&pDirectSoundBuf,NULL)))
    return false;          //取得DirectSoundBuffer缓冲区接口对象
if(FAILED(pDirectSoundBuf->QueryInterface(IID_IDirectSoundBuffer8,&pDirectSo
undBuf8)))   //取得IDirectSoundBuffer缓冲区接口对象
(pDirectSoundBuf->Release();
return false;)
```

3. 加载声音数据到次缓冲区

创建了次缓冲区之后，就要将 wav 文件中的声音数据加载到次缓冲区中进行播放，等于对缓冲区进行写入的操作。因此，必须要首先利用 IDirectSoundBuffer8 接口提供的 Lock 函数锁定缓冲区，待完成加载的操作后再解除锁定，进行写入操作。下面是 Lock 函数的使用原型。

```
HRESULT Lock(
    DWORD dwOffset,              //锁定开始位置
    DWORD dwBytes,               //锁定字节数
    LPVOID *ppvAudioPtr1,        //返回第一个位置指针
    LPDWORD pdwAudioBytes1,      //锁定第一个缓冲区块的大小
    LPVOID *ppvAudioPtr2,        //返回第二个位置指针
    LPDWORD pdwAudioBytes2,      //锁定第二个缓冲区块的大小
    DWORD dwFlags                //锁定标志
);
```

下面是调用 Lock 方法锁定次缓冲区时，其中每个参数设定的说明，如表 4-2 所示。

表 4-2 每个参数设定的说明

参数	说明
0	要锁定缓冲区的开始位置，0 代表从最前面开始
size	要锁定缓冲区的大小
&pAudio	执行缓冲区的指针
&bytesAudio	缓冲区中要存入数据的长度
NULL	第二个指向缓冲区的指针，设定为 NULL 时表示不划分第二个区块
NULL	缓冲区中第二个区块要存入数据的长度
NULL	设定标志，此处设定为 NULL

IDirectSoundBuffer 的缓冲区采用环形的形式进行读和写，以便于声音数据可以随机地写入和读取。假设缓冲区总共有 1 000 个字节，但需要全部锁定，锁定后返回的第一个读写位置指针 ppvAudioPtr1 指向当前的位置 200，可以看出这个块只有 800 个字节。然后，再从尾部绕回缓冲区的头部，锁定 100 个字节，此时返回的第二个读写指针必然指向缓冲区的开始位置 0 处，图 4-3 可以帮助读者理解。

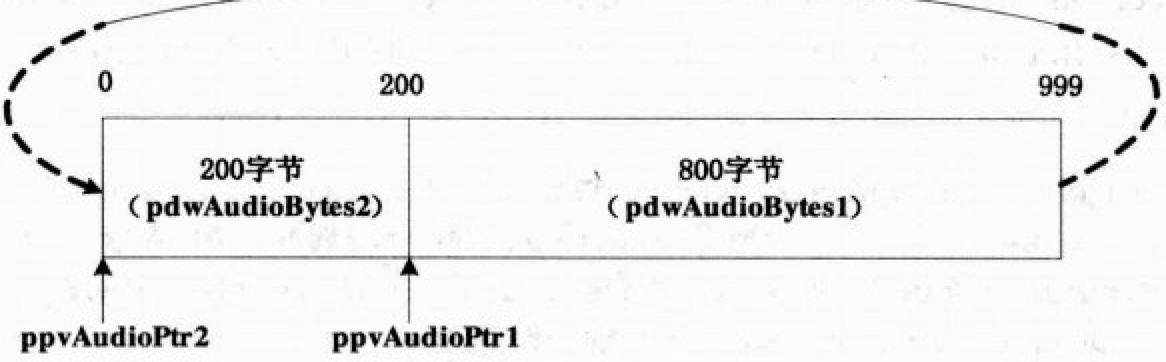


图 4-3 Lock 函数的缓冲区锁定

锁定标志 dwFlags 参数默认取值 DSBLOCK_FROMWRITECURSOR=0，表示缓冲区当前读写位置作为第一个锁定块的返回指针，这个时候 dwOffset 起始位置参数将被忽略。此参数的另一个取值为 DSBLOCK_ENTIREBUFFER，表示锁定整个缓冲区，而 dwBytes 锁定字节数将被忽略。

当缓冲区锁定后，就可以将 wav 文件的 data 块后面的实际声音数据分为两次写入锁定缓冲区块，如下面代码所示。

```
//锁定缓冲区
BYTE *pSoundPtr1=NULL; //必须为NULL
BYTE *pSoundPtr2=NULL; //必须为NULL
DWORD dwSoundSize1,dwSoundSize2; //定义写位置指针
if(FAILED(pDirectSoundBuf8->Lock(0,pWaveHeader->DataSize, //锁存全部缓冲区
                                  (void**)&pSoundPtr1,&dwSoundSize1,
                                  (void**)&pSoundPtr2,&dwSoundSize2,0)))
    return false;
//读数据到缓冲区
fread(pSoundPtr1,1,dwSoundSize1,fp);    //写入第一个缓冲区锁定块
if(pSoundSize2 != NULL)
fread(pSoundPtr2,1,dwSoundSize2,fp);    //写入第二个缓冲区锁定块
```

当声音数据写入次缓冲区之后，需要调用 IDirectSoundBuffer8 接口的 Unlock 函数对缓冲区进行解锁。下面是 Unlock 函数的原型。

```
HRESULT Unlock(
    LPVOID pvAudioPtr1,       //第一个缓冲区锁定块指针
LPVOID pvAudioPtr2,           //第一个缓冲区锁定块大小
    DWORD dwAudioBytes1,      //第二个缓冲区锁定块指针
    DWORD dwAudioBytes2,      //第二个缓冲区锁定块大小
);
```

而解锁 UnLock 函数的使用也十分简单，只需要传入两个锁定块的指针和大小即可，如下面代码所示。

```
pDirectSoundBuf8->Unlock(pSoundPtr1,dwSoundSize1,pSoundPtr2,dwSoundSize2);
 //对缓冲区解锁操作
```

以上完成了 DirectSound 程序的基本结构，而且也在次缓冲区加载了所要使用的声音文件数据。下面将讨论如何运用次缓冲区中的数据来产生所要的声音效果。

4.3.3 调用方法播放声音

当把声音数据加载到次缓冲区之后，就要调用 IDirectSoundBuffer8 接口提供的 Play 函数来播放声音了。Play 函数实际上是将次缓冲区的声音数据送到混声器中，并和其他正在播放的声音进行混合，最后输出到主缓冲区自动播放。下面是它的原型。

```
HRESULT Play(
    DWORD dwReserved1,      //必须为0
    DWORD dwPriority,       //优先权，一般为0
    DWORD dwFlags           //播放方式标志位
);
```

参数 dwPriority 是声音混音时的优先权，范围是 0～0xFFFFFFFF，0 是最低权限。如果创建缓冲区时没有使用 DSBCAPS_LOCDEFER 标志，则优先权必须设置为 0。

参数 dwFlags 指明播放的方式，其说明如表 4-3 所示。

表 4-3 参数 dwFlags 的说明

参数值	说明
DSBPLAY_LOOPING	循环播放，直到调用 Stop 函数停止。主缓冲区播放必须使用此参数
DSBPLAY_LOCHARDWARE	只能通过硬件中的缓冲区进行播放，否则调用失败
DSBPLAY_LOCSOFTWARE	只能在软件缓冲区中进行播放，不能组合使用上面的 DSBPLAY_LOCHARDWARE 参数

如果要循环播放声音，就要使用 DSBPLAY_LOOPING（=1）参数值。如果只播放一次就停止，则将 dwFlags 参数值设为 0，DirectX 并没有为它提供一个标志宏，相当于 dwFlags 取值 true/false 时，声音循环播放或只播放一次。

一般设置前两个参数的值为 0，最后一个参数取 true 或者 false 来播放缓冲区的声音，循环播放缓冲区中的声音的代码如下。

```
if(FAILD(pDirectSoundBuf8->Play(0,0,true)))  //利用Play()函数播放声音
    return false;
```

声音的播放位置，可以通过 IDirectSoundBuffer8 接口的 SetCurrentPosition 函数来设置，此函数的原型为：

```
HRESULT SetCurrentPosition( DWORD dwNewPosition  //距离缓冲区的偏移位置
);
```

下面是将声音移到最前面的位置，然后再播放。

```
pDirectSoundBuf8->SetCurrentPosition(0);     //设置声音的播放位置
if(FAILED(pDirectSoundBuf8->Play(0,0,false))) //播放一次
    return false;
```

4.3.4 结束播放

当声音需要中途停止播放时，可以通过调用IDirectSoundBuffer8接口的Stop函数来实现。函数的使用形式很简单，不需要对齐传递任何参数。具体代码如下所示。

```
HRESULT Stop();
```

下面的代码将停止声音的播放。

```
if(FAILED(pDirectSoundBuf8->Stop()))  //停止播放函数
    return false;
```

然后介绍一下IDirectSoundBuffer8接口的音量控制函数SetVolume()和调整播放声道的函数SetPan()。

首先看如何调整次缓冲区的音量大小，控制音量函数SetVolume()的使用语法如下。

```
HRESULT SetVolume(
LONG lVolume    //音量大小
);
```

此函数传入的参数1Volume是一个Long类型的音量大小值，介于0～-10 000之间的数值，负值越小表示声音越小。而SetVolume函数并没有放大音量的功能，只能把音量往下调。若需要音量较大的声音文件，则必须自行使用音效编辑软件来制作，而声音文件加载到次缓冲区时，便已经是最大的音量了。0实际上为最大音量，没有任何衰减，而-10 000实际的音量最小，无声音，衰减程度最大。

而控制播放声道比较常见的就是控制左、右声道的声音大小，利用不断连续地切换左、右声道声音的播放产生立体音的效果，DirectSound中调整播放声道的函数是SetPan()函数，其语法为：

```
HRESULT SetPan(
   LONG lPan  //设置声道
);
```

其中所传入的参数1Pan也是一个Long类型的值，其值介于-10 000～10 000之间，负值越小表示左声道的声音越大，正值越大则表示右声道的声音越大，为0时表示两声道的音量相等。

通过上面两种方法的介绍，会发现在DirectSound中要控制音量的大小或者是声道声音播放的强弱其实是很简单的。

4.4 系统效果演示

在4.3节中介绍了播放声音的基本实现原理，下面是具体的实现步骤。

4.4.1 新建基于对话框的程序框架

STEP 01 启动 Microsoft Visual Studio 2005，选择“文件”→“新建”→“项目”命令，项目名称为 Sound，如图 4-4 所示。单击“确定”按钮，进入下一步。

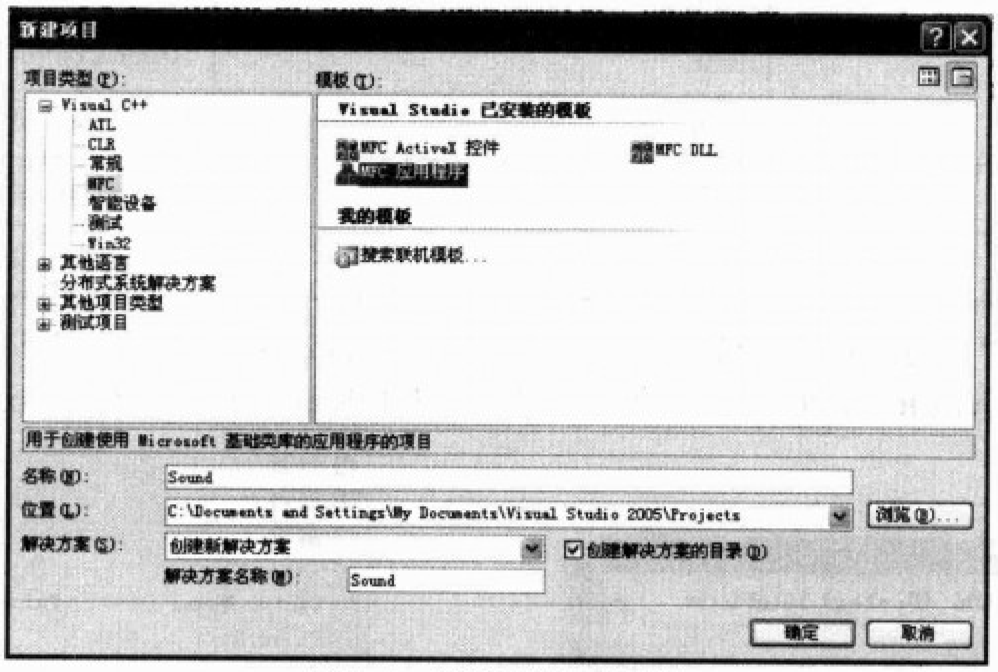


图 4-4 新建项目对话框

STEP 02 在其中选择“基于对话框”单选按钮，并在“资源语言”下拉列表中选择“中文”选项，单击“完成”按钮就可生成相应的程序框架。这样就创建了一个基于对话框的 MFC 应用程序，如图 4-5 所示。

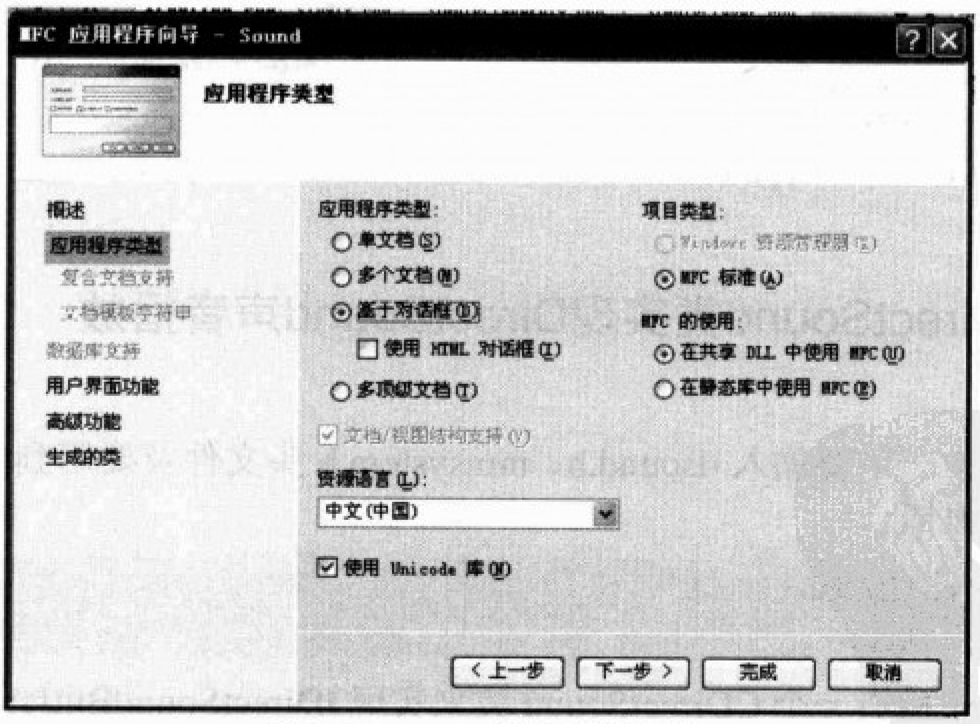


图 4-5 MFC 应用程序向导

4.4.2 添加控件及变量

新建一个基于对话框的程序框架以后，然后在对话框上按表 4-4 放置相应的控件。

表 4-4 控件属性

ID	标题	控件类型
IDC_OPEN	打开	Button Control
IDC_PLAY	播放	Button Control
IDC_PAUSE	暂停	Button Control
IDC_STOP	停止	Button Control
IDC_FAST	快进	Button Control
IDC_NORMAL	正常	Button Control
IDC_SLOW	减速	Button Control

在对话框的“Caption”中输入“自己的音频播放器”，如图 4-6 所示。“ID”设为“IDD_SOUND_DIALOG”，如图 4-7 所示。设置控件的标题属性（Caption）和 ID 与之相同。

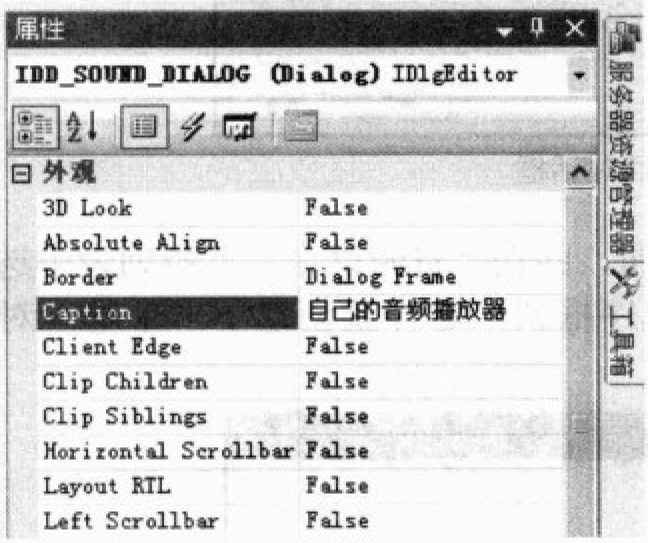


图 4-6 设置 Caption

图 4-7 设置 ID

4.4.3 封装CDirectSound类实现DirectSound声音播放

该类在头文件中一定要加入 dsound.h、mmsystem.h 头文件及要用到的头文件，如果未加入，则在编译时会报错。

```
#include <mmsystem.h>
#include <dsound.h>
```

系统实现时首先构造一个 CDirectSound 类来实现 IDirectSoundBuffer 接口的封装和管理。此接口实现了次缓冲区的创建、释放、声音数据的填充、播放、停止等功能，下面设计的代码分别是 CDirectSound 类的构造函数、析构函数、数据获取和填充函数、播放函数、停止播放函数、暂停函数和继续播放函数等，还定义了一些函数成员变量和标识符。

封装 CDirectSound 类的程序流程如图 4-8 所示。

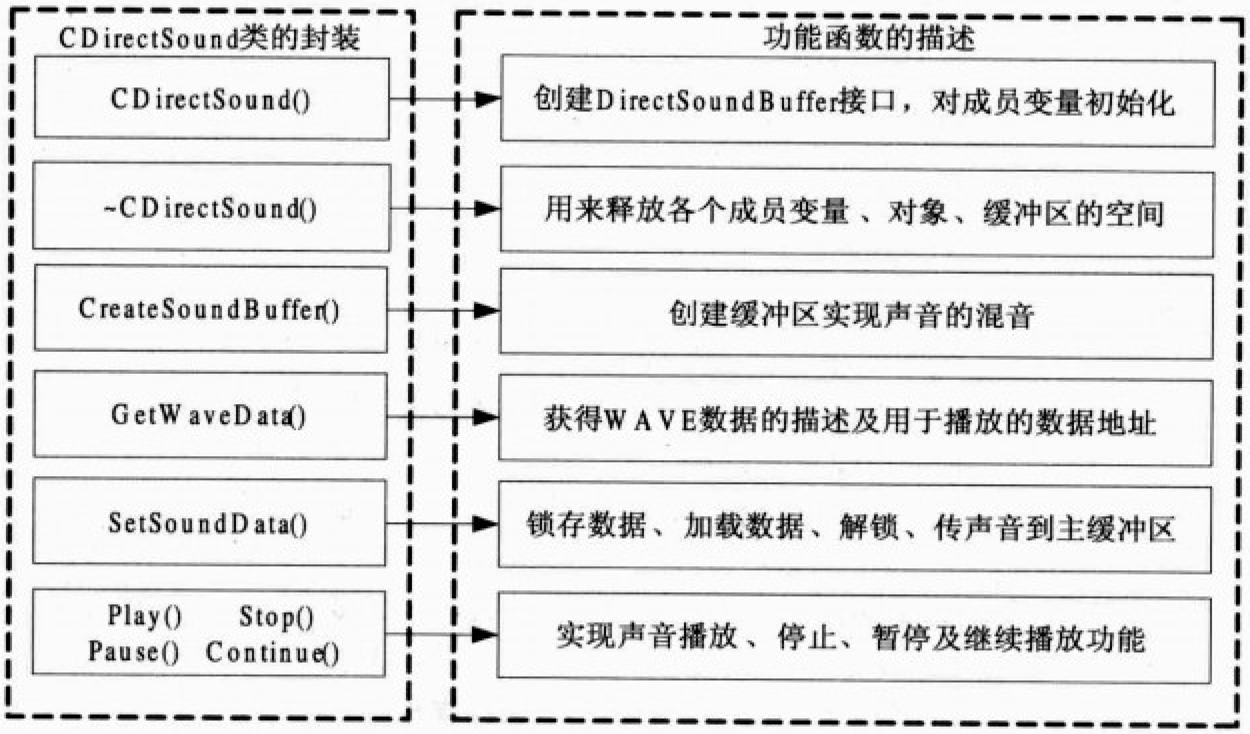


图 4-8 类 CDirectSound 的封装流程

1. CDirectSound 类的封装

定义 CDirectSound 类，用于完成实体的操作。

```
class __declspec(dllexport) CDirectSound
{
public:
 CDirectSound();           //构造函数
 ~CDirectSound();          //析构函数
 BOOL  WriteToBuf();
 int    WriteDataToBuf(BYTE* pBuf,DWORD dwLen); //写数据到缓冲区
 BOOL  CreateDSound(AUDIO_CONFIG* WaveHead,DWORD dwHeadLen);
                                              //创建缓冲区
 BOOL  SetPlaySpeed(float fSpeed);  //设置播放速度函数
public:
 void  Play(INT nTime = 0, BOOL bLoop = TRUE);
 void   Stop();        //定义停止函数
 void    Pause();      //定义暂停函数
 void    Continue();  //继续播放函数
protected:
 BOOL    IsValid() const;
 BOOL  GetWaveData(void * pHeader, WAVEFORMATEX * & pWaveHeader,
                  DWORD & dwHeadSize); //获取声音数据
 BOOL  GetWaveData(void * pRes, WAVEFORMATEX * & pWaveHeader,
                         void * & pbWaveData, DWORD & cbWaveSize);
 BOOL  SetSoundData(void * pSoundData,DWORD SoundSize); //填充声音数据
 BOOL  SetSoundData(DWORD Offset,void * pSoundData, DWORD dwSoundSize);
 BOOL  CreateSoundBuffer(WAVEFORMATEX * pcmwf);    //创建缓冲区
private:
```

```
DSBPOSITIONNOTIFY   DSNotify[16];
AUDIO_CONFIG        m_config;
float               m_fRate;
int                 m_rate;
    int                 i;
    CString             file[255];
    CString             filemp;
DWORD               m_nSamplesPerSec;    //采样率
DWORD               m_nAvgBytesPerSec;   //每秒采样字节数
DWORD               m_dwTotalSize;
DWORD               m_dwHeadLen;
DWORD               m_dwBufNum;
DWORD               m_dwBufLen;          //缓冲区长度
DWORD               m_dwBufSize;         //缓冲区大小
DWORD               m_dwOffset;          //偏移量
DWORD               m_dwLen;
DWORD               m_dwTheSound;
HANDLE              notify_events[16];   //通知事件
HANDLE              m_hPlayThread;       //播放线程
HANDLE              m_sysncObj;
HANDLE              m_hEvent;
HWND                m_hWnd;
BYTE*               m_pTheSound;
BOOL                m_bStop;             //停止播放函数成员变量
BOOL                m_bFir;
BOOL                m_bEnabled;          //是否使能
BOOL                m_bPlay;             //播放函数成员变量
  BOOL                m_play_flag,m_pause_flag;  //播放和暂停标识符
static LPDIRECTSOUNDNOTIFY  m_pDsNotify;
static LPDIRECTSOUNDBUFFER  m_pDsb;
static LPDIRECTSOUND m_lpDirectSound;
static DWORD          m_dwInstances;
};
```

2. 创建主缓冲区

```
BOOL  CDirectSound::CreateDSound(AUDIO_CONFIG* WaveHead,
DWORD dwHeadLen) //定义创建主缓冲区函数
{
if(m_config.nSamplesPerSec == NULL)
{
    m_fRate = 1.0;
    m_config = *WaveHead;                              //WAVE头文件指针
    m_nAvgBytesPerSec = m_config.nAvgBytesPerSec;      //设置平均每秒采样字节数
    m_nSamplesPerSec  = m_config.nSamplesPerSec;       //采样率
}
m_dwBufLen = dwHeadLen ;
```

```
    if(m_lpDirectSound)
    {
         return TRUE;
    }
    HRESULT hRes = DS_OK;
    short nRes = 0;
    do {
              if( nRes )
                   ::Sleep(500);  //延时0.5秒
              hRes = ::DirectSoundCreate(0, &m_lpDirectSound, 0);++nRes;}
    while( nRes < 10 && (hRes == DSERR_ALLOCATED ||
    hRes == DSERR_NODRIVER) );
    if( hRes != DS_OK )
         return FALSE;
    m_lpDirectSound->SetCooperativeLevel(/*m_hWnd*/::GetDesktopWindow(),
DSSCL_NORMAL);    //设置协作级别，设为DSSCL_NORMAL
    m_rate = WaveHead->nAvgBytesPerSec;
    if(!CreateSoundBuffer((WAVEFORMATEX*)(&m_config)))
    //在主缓冲区设置格式类型
         return FALSE;
```

通用 CreateSoundBuffer()函数从声卡内存中进行主缓冲区的创建分配。在主缓冲区中设置采样率、协调级别及格式类型。主缓冲区的目的是实现声音的混音，混音后，数据从主缓冲区传递到硬件，然后转化为声音。

3. CDirectSound::GetWaveData()获得 WAVE 数据

获得 WAVE 数据文件的步骤如下。

STEP 01 打开文件。

STEP 02 确认是否为 RIFF 文件，并且类型为 WAVE。

STEP 03 寻找 fmt 块，获得文件的格式。

STEP 04 寻找 data 块，获得文件的内容。

STEP 05 关闭文件。

CDirectSound::GetWaveData()从 WAVE 数据块中得到其描述的结构与可用于播放的数据地址，代码如下。

```
BOOL CDirectSound::GetWaveData(void*pRes,WAVEFORMATEX *& pWaveHeader,
     void*& pbWaveData, DWORD & cbWaveSize ) //获取WAVE数据
{
pWaveHeader = 0;      //头文件指针
pbWaveData = 0;       //WAVE数据指针
cbWaveSize = 0;       //附加的数据大小
DWORD * pdw = (DWORD *)pRes;
DWORD dwRiff = *pdw++;
DWORD dwLength = *pdw++;
```

```
DWORD dwType = *pdw++;
if( dwRiff != mmioFOURCC('R', 'I', 'F', 'F') )  //检查是否为WAV文件
      return FALSE;                    // 不是RIFF
if( dwType != mmioFOURCC('W', 'A', 'V', 'E') )
      return FALSE;                    // 不是WAV
DWORD * pdwEnd = (DWORD *)((BYTE *)pdw + 1000-4);
while( pdw < pdwEnd ) {dwType = *pdw++;dwLength = *pdw++;switch( dwType )
{                                      //搜寻是否有fmt和data区块
      case mmioFOURCC('f', 'm', 't', ' '):
      if( !pWaveHeader ) {
                 if( dwLength < sizeof(WAVEFORMAT) )
                 return FALSE;     //不是 WAV
                        pWaveHeader = (WAVEFORMATEX *)pdw;
                 if( pbWaveData && cbWaveSize )
                           return TRUE;
            }
            break;
      case mmioFOURCC('d', 'a', 't', 'a'):
                 pbWaveData = LPVOID(pdw);
                 cbWaveSize = dwLength;  //附加的数据大小
                 if( pWaveHeader )
                       return TRUE;
                 break;
 }
 pdw = (DWORD *)((BYTE *)pdw + ((dwLength+1)&~1));
}
return FALSE;
}
```

这段程序会判断所打开的文件是否为 WAVE 类型（RIFF 区块中必须有 WAVE 字符串），并搜寻其中是否有 fmt 和 data 区块。

4. 创建次缓冲区

在得到声音文件的格式和数据大小之后，就可以来建立存储声音的次缓冲区了。CDirectSound::CreateSoundBuffer()调用 IDirectSound::CreateSoundBuffer 方法创建一个基本的次缓冲区。代码如下。

```
BOOL CDirectSound::CreateSoundBuffer(WAVEFORMATEX * pcmwf)
{
DSBUFFERDESC dsbdesc;                     //创建DSBUFFERDESC结构体
memset(&dsbdesc, 0, sizeof(DSBUFFERDESC));
dsbdesc.dwSize = sizeof(DSBUFFERDESC);    //获取结构体大小
dsbdesc.dwFlags = dsbdesc.dwFlags =       //表示主缓冲区
   DSBCAPS_GETCURRENTPOSITION2
   |DSBCAPS_CTRLPOSITIONNOTIFY|DSBCAPS_GLOBALFOCUS;
   dsbdesc.dwBufferBytes =m_dwBufNum*m_dwBufLen;
```

```
    dsbdesc.lpwfxFormat = pcmwf;                        //创建缓冲区
     HRESULT hRes;
    if(DS_OK!= (hRes = m_lpDirectSound->CreateSoundBuffer(&dsbdesc, &m_pDsb, 0)) ) {
          DSError(hRes);
          m_pDsb = 0;
          return FALSE;
    }
    for (int i=0;i<m_dwBufNum;i++)
    {
          // 创建缓冲区通知，自动重设事件
          m_hEvent=CreateEvent( NULL, FALSE, FALSE, NULL ); //创建事件
          DSNotify[i].hEventNotify=m_hEvent;
          DSNotify[i].dwOffset=i*(m_dwBufLen);     //通知点数
    }
    hRes=m_pDsb->QueryInterface(IID_IDirectSoundNotify,(void**)&m_pDsNotify);
    if (FAILED(hRes))                                   //成功
    {
                return FALSE;
    }
    // 获得通知位置
    hRes=m_pDsNotify->SetNotificationPositions(m_dwBufNum,DSNotify);
    if (FAILED(hRes))                                   //成功
    {
                return FALSE;
    }
    for (int n=0;n< m_dwBufNum;n++)
    {
          notify_events[n]=DSNotify[n].hEventNotify;
          ResetEvent(notify_events[n]);            //重设事件
    }
    return TRUE;
    }
```

首先创建描述缓冲区的结构体，定义结构体中的参数变量，然后创建次缓冲区，设置缓冲区的通知点，以便在这些通知点上进行相应的操作，但是在播放过程中不能调用该函数。

5. CDirectSound::SetSoundData()播放声音

播放声音的过程有4步。

STEP 01 锁定缓冲区（IDirectSoundBuffer::Lock）。

STEP 02 加载数据到缓冲区。

STEP 03 解除锁定（IDirectSoundBuffer::Unlock）。

STEP 04 将声音送到主缓冲区，并输出播放（IDirectSound:;Play）。

CDirectSound::SetSoundData()具体实现代码如下。

```
BOOL CDirectSound::SetSoundData(void * pSoundData,DWORD SoundSize)
{
LPVOID lpvPtr1;                          //定义写指针
DWORD dwBytes1;
if (!pSoundData) return FALSE;    //获得写指针
HRESULT  hr=m_pDsb->Lock(0,SoundSize,&lpvPtr1,
                          &dwBytes1,0,0, SBLOCK_ENTIREBUFFER);
    //如果DSERR_BUFFERLOST被返回，则恢复和重新锁定
if(DSERR_BUFFERLOST == hr) {
     m_pDsb->Restore();              //重新恢复
hr=m_pDsb->Lock(0,SoundSize,&lpvPtr1,&dwBytes1,0,0, DSBLOCK_ENTIREBUFFER);
}                                        //重新锁存
if(DS_OK == hr) {
     //写到指针处
     ::CopyMemory(lpvPtr1, pSoundData, SoundSize);
     //加载数据到DirectSound
     hr = m_pDsb->Unlock(lpvPtr1, dwBytes1, 0, 0);
     if(DS_OK == hr)
            return TRUE;
}     //锁存，解锁，或者恢复失败
return FALSE;
}
```

首先要定义写指针，其次要获得写指针的位置。锁存声音缓冲区，如果缓冲区丢失，则需要重新恢复和锁存，然后在写指针处加载声音数据，最后进行解锁，准备播放声音。

6. 声音的播放、停止等操作函数

下面讲解声音的播放、停止等操作函数，以实现通过文件的方式来打开播放声音文件，具体实现的相关代码如下。

1）打开操作的实现

```
BOOL CDirectSound::Open()
{
   CString str,
char lpszFilter[]="MP3 File(*.mp3)|*.mp3|"    //打开文件对话框，选择要播放声音的格式
                  "Wave File(*.wav)|*.wav|"
                  "All Files(*.*)|*.*|"
CFileDialog dlg(TRUE,NULL,NULL,OFN_HINDEREADONLY, lpszFilter)
                                              //创建对话框对象
POSITION pst=str.GetStartPosition();          //获取开始位置
if(dlg.Domodal() == IDOK)                     //打开模式对话框
{
     CString str=dlg.GetPathName();           //获取路径
     While(pst!=NULL)
     {
```

```
            file[i]=str.GetNextPathName(pst);       //若获取路径失败，则获取下一个文件路径
            if(filemp =="")
            {
                   filemp=file[i];
                   m_play_flag=false;               //设置播放标志
                   m_pause_flag=false;
             }
        }
}
```

打开操作，则首先打开文件对话框，选择要播放的声音格式，然后创建对话框对象，获取播放路径、设置播放标志等操作，为播放做好准备。

2）播放操作的实现

```
void CDirectSound::Play(INT nTime, BOOL bLoop)
{
 if( ! IsValid()
        return;
 DWORD dwPlayCursor;                         //定义播放指针
 DWORD dwWriteCursor;                        //定义写指针
 m_pDsb->GetCurrentPosition(&dwPlayCursor, &dwWriteCursor);
//获取播放的当前位置
 int offByte = nTime*m_rate/1000;      //获取偏移量
 if(dwPlayCursor >0)
 {
       if( dwPlayCursor > m_dwBufLen* m_dwBufNum )
       dwPlayCursor = m_dwBufLen* m_dwBufNum;
 }
 if(dwPlayCursor > offByte)
 {
       if(dwPlayCursor  < -offByte)
       {
             dwPlayCursor = 0;
       }
       else
       {
             dwPlayCursor += offByte;  //播放指针加上偏移量，获得播放位置
       }
 }
 else
       dwPlayCursor = m_dwBufLen* m_dwBufNum;
       m_pDsb->SetCurrentPosition(dwPlayCursor);  //设置当前播放位置
 if( DSERR_BUFFERLOST == m_pDsb->Play(0, 0, 1) )
 {
       SetSoundData(m_pTheSound, m_dwBufLen);      //获得声音数据
```

```
        //重试
        m_pDsb->Play(0, 0, 1);          //播放声音文件
    }
}
```

此程序说明了播放的操作流程。首先定义播放指针，获取当前播放的位置，设置播放的位置，加载声音数据，最后播放声音文件。

3）暂停播放操作的实现

```
void CDirectSound::Pause()
{
    if( IsValid() )
            m_pDsb->Stop();             //调用停止播放函数，停止播放
}
```

此代码调用 CDirectSound 类中的 Pause()函数，实现暂停播放的功能。

4）停止操作的实现

```
void CDirectSound::Stop()
{
m_bStop = 1;                            //运行停止播放
if(m_hPlayThread)
{
    WaitForSingleObject(m_hPlayThread,2000);
    CloseHandle(m_hPlayThread);         //关闭播放线程
    m_hPlayThread = NULL;               //播放线程为NULL
}
if( IsValid() )
    m_pDsb->Stop();                     //停止播放
for (int i=0;i<m_dwBufNum;i++)
{
    ResetEvent(notify_events[i]);       //自动重置事件
    CloseHandle(notify_events[i]);      //=m_hEvent;
}
```

此代码调用 DirectSound 类中的 Stop()函数，实现停止播放功能。首先设置运行停止播放标志，然后关闭线程，最后停止声音播放。

4.4.4 与对话框界面相关操作实现

上面详细介绍了 CDirectSound 类的封装，此类具有管理声音播放的功能，可以实现 DirectSound 对象的创建，缓冲区的创建、删除等，同时在此类中定义了管理声音的相关函数，并详解分析了这些函数的代码功能。下面就介绍播放器对话框界面上的具体按钮功能的实现。设置和实现对话框界面的按钮功能，具体的流程如图 4-9 所示，管理声音的相关函数如图 4-10 所示。

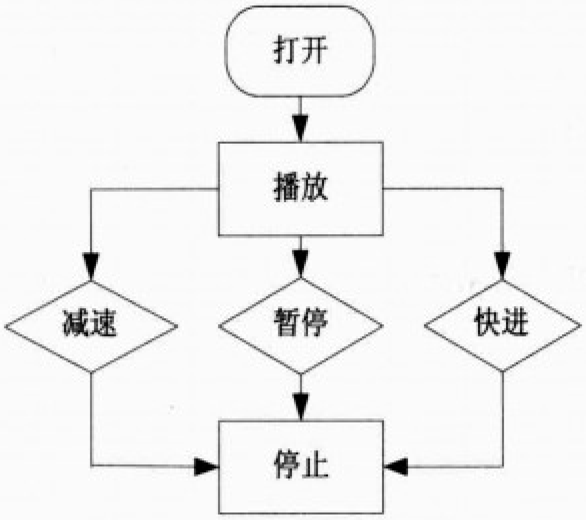


图 4-9 播放器的实现流程

图 4-10 管理声音的相关函数

与对话框界面相关操作实现如下。

1. 打开功能的实现

```
void CSoundDlg::OnOpen()
{
CString m_Path, strIndex, strTruthMusicTime; //定义播放路径、播放指针、播放时间
strIndex.Format("%d: ", num+1);
      _TCHAR strFileFilter[] = "MP3 File(*.mp3)|*.mp3|"
                               "WMA File(*.wma)|*.wma||"; //打开指定的声音格式
CFileDialog Dlg(TRUE,NULL ,NULL, OFN_HIDEREADONLY,strFileFilter);
if(Dlg.DoModal() == IDOK)                       //打开模式对话框
{
      m_Path = Dlg.GetPathName();               //获取路径
      OnPlay(m_Path, strIndex);
      OutputItem(m_Path);
      index = num - 1;
      m_AddList.SetSelectionMark( index );      //加载要播放的曲目
}
}
```

当单击“打开”按钮时，首先要获取播放的路径、播放的指针、播放的声音的真实时间等，然后打开所指定的声音格式，最后加载所要播放的曲目。

2. 播放功能的实现

```
void CSoundDlg::OnBnClickedPlay()
{
 AUDIO_CONFIG WaveHead;                 //调用声音头文件
 WaveHead.wFormatTag = 1;
 WaveHead.nChannels = 2;                //通道数
 WaveHead.nSamplesPerSec = 44100;       //采样率
```

```
    WaveHead.nAvgBytesPerSec = 176400;   //平均每秒采样字节数
    WaveHead.nBlockAlign = 4;            //块对齐大小
    WaveHead.wBitsPerSample = 16;        //位数
    m_bStop = 0;
    unsigned int dwReadID;
    m_bContine = 1;
    fp = fopen("a.wav","rb");            //打开.wav文件
    BYTE buf[1025];
    int nlen = fread(buf,1,1024,fp);     //读文件长度
    if(nlen < 1024)                      //若长度小于1024，则关闭
    {
       fclose(fp);
       fp = NULL;
       return ;
    }
    else
    {
       m_sndSound1->CreateDSound(&WaveHead,8000); //否则创建，准备播放
       fseek(fp,44,0);
    }
       m_hThread = (HANDLE)_beginthreadex(0,0,FileReadProc,this,0,&dwReadID);
                               //系统开始创建线程
    Sleep(200);
    m_sndSound1->Play();     //播放WAV 声音文件
}
```

当单击“播放”按钮时，首先读取WAV文件的声音格式数据，通过WAV文件头的结构体设置播放声音的各项参数，随即打开WAV文件，创建线程进行播放。

3. 暂停功能的实现

```
void CSoundDlg::OnBnClickedPause()
{
if(m_bContine)
{
     m_sndSound1->Pause();          //暂停播放
     m_bContine = 0;
}
else
{
     m_sndSound1->Continue();     //否则继续播放
     m_bContine = 1;
}
}
```

当单击界面上的“暂停”按钮时，播放器会暂停播放声音文件，否则播放器会继续播放声音文件。

4．停止功能的实现

```
void CSoundDlg::OnStop()
{
m_bStop = 1;
if(m_hThread)
{
    WaitForSingleObject(m_hThread,2000);
    CloseHandle(m_hThread);      //关闭线程
    m_hThread = NULL;
}
m_sndSound1->Stop();             //停止播放
}
```

当单击“停止”按钮时，关闭线程，调用 Stop()函数，播放随即停止。

5．快进功能的实现

```
void CSoundDlg::OnFast()
{
// TODO：在此添加控件通知处理程序代码
m_sndSound1->SetPlaySpeed(2.0);  //设置快进速度
}
```

当单击“快进”按钮时，通过 SetPlaySpeed()函数可以设置播放的速度，此程序把播放速度提快 2.0 秒。

6．减速功能的实现

```
void CSoundDlg::OnSlow()
{
    //TODO：在此添加控件通知处理程序代码
    m_sndSound1->SetPlaySpeed(0.5); //设置所要减慢的速度
```

当单击“减速”按钮时，通过调用 SetPlaySpeed()函数来设置播放速度，这里播放速度减慢了 0.5 秒。

源代码中其余的部分都是系统自动生成的，此处就不一一列举了。在本案例需要注意以下几点。

- 采用面向对象的思路来分析和实现代码，这样才能编写高质量和强移植性的代码。
- 可以根据实际需要采用不同层次的 Windows API 函数来完成所需功能。

4.4.5 程序运行结果

音频播放器的实现代码在上面已经详细地分析和设计了，最后就是要编译和运行程序，来实现音频播放的目的。

1. 编译与运行

首先，选择“生成”→“编译”选项，系统将会编译“DirectSound.cpp”文件；接着选择“生成”→“生成解决方案”选项，系统将会完成链接并生成该项目的可执行文件。

当然，如果不想一个个地编译源文件的话，也可以一开始就选择“生成”菜单下的“生成解决方案”命令，系统将会同时完成编译和链接，生成该项目的可执行文件。

在编译运行程序时，界面的下方“输出”窗口将会显示许多的信息，包括编译的状态、是否有错误、编译链接的结果等，有必要的时候可以从此窗口中获得更多的信息。更多的调试方法在第3章已介绍过。

2. 运行程序

单击“调试”→“开始执行（不调试）”按钮，运行本程序，程序运行结果如图4-11所示。单击“打开”按钮，选择曲目，再单击“播放”按钮，即可播放。单击“暂停”按钮，播放立刻暂停，其余功能单击对应的按钮均可实现。

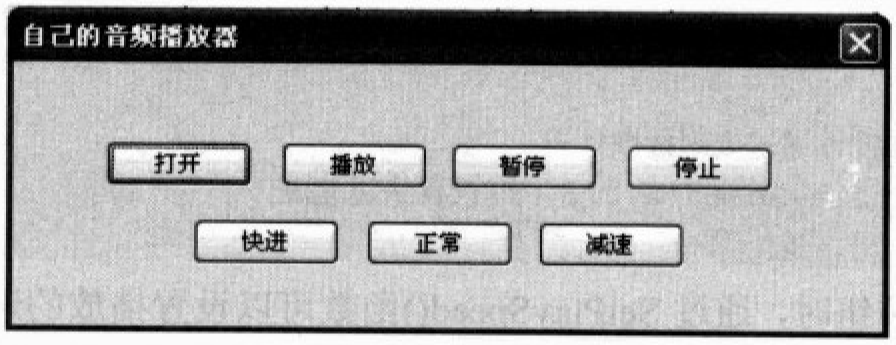


图4-11 程序运行界面

4.5 DirectSound开发常见问题

用DirectSound开发音频应用程序，它的优点已经不言而喻，特别是在游戏开发中用得很多，并且提供了强大的开发工具。但是在开发过程中往往会碰到一些问题，下面列举了一些常见的问题和解决方法。

1）*在DirectSound开发前要注意哪些*

在DirectSound编程前，我们开发的DirectSound类在编译时需要用到DirectX 9.0开发包的dsound.h头文件，所以在类的头文件前加入声明。

```
#include <mmsystem.h>
#include <dsound.h>
```

这些头文件是DirectSound程序所要用到的。在程序链接时，DirectSound要用到Dxguid.lib和Dsound.lib的库文件，这也是必须的，详细设置在第3章中已介绍过，这里不再赘述。

2）*想要最大限度地发挥硬件的增强功能，应如何了解设备的特性*

前面讲到DirectSound有一个特性，就是当硬件空闲时可自动启用硬件加速。那么为了充分发挥硬件的增强功能，就需要先了解设备的特性，这个可以通过IDirectSound::GetCaps

方法来获取。函数原型如下所示。

```
DSCAPS dscaps;
dscaps.dwsize = sizeof(DSCAPS);
HRESULT hr = lpDirectSound->lpVtbl->GetCaps(lpDirectSound,&descaps);
```

DSCAPS 可以接收关于声音设备的性能和资源的信息。但是要注意，初始化该结构中 dwSize 成员是调用此方法之前所必须的。

另外，也可以查看和设置扬声器的配置，从而可以腾出声音存储器来尽可能获得最大的备用空间。

3）为什么在很多应用程序中，对主缓冲区的直接操作很少

首先次缓冲区不支持混音等特殊效果，所以就需要直接操作主缓冲区。但是当获准对主缓冲区进行写操作时，其他的特性就将失效，结果硬件加速混音就失去作用了。如果要对主缓冲区进行操作，可调用 IDirectSound::CreateSoundBuffer 方法来设置 DSBUFFERDESC 结构中的 DSBCAPS_PRIMARYBUFFER 标识符，同时协作级别必须设为写主缓冲区协作级别。

4.6 本章小结

本章主要是用 DirectX 中的 DirectSound 组件设计了一个简单的音频播放器。实现了基本的播放、停止等功能。详细介绍了 DirectSound 编程的基本方法，涉及缓冲区的创建、声音的播放等的使用方法，DirectSound 中还有很多特殊音效，如回声和齐声等，读者可以在编程时加入，从而达到满意的音效效果。同时也学习了 Visual 2005 基本的控件操作方法。音频播放器界面比较简单，在此基础上读者可以试着创造更加复杂、更加美观的窗口。

第5章

语音3D特效器制作

实现3D音效是DirectSound的一项很重要的功能。DirectSound对真实感声音模拟提供了很好的支持，是目前在3D声音生成领域非常优秀的开发接口。通过建立发声者和听者的位置、声音速度和声音锥区域等信息，可以实现所期待的3D声音特性。本文将通过一个案例来具体介绍DirectSound是如何实现具有震撼力的3D声音的。

本章重点

- 系统设计和分析
- DirectSound 3D 缓冲区
- DirectSound 3D 虚拟听众
- 系统效果演示
- DirectSound 3D 开发常见问题

5.1 系统设计和分析

本案例设计制作了一个语音3D特效器，通过DirectX SDK的DirectSound中的3D音效来实现具有震撼力的3D音效。可以实现最大距离、最小距离、多普勒因子和滚降衰减等音效，同时也可以模拟出声源的运动。通过本案例可以从中学习到如何利用DirectSound相应的接口来实现3D声音效果。

DirectSound 3D可以说是DirectSound的精华所在，它利用了声音大小的比例调整和多普勒效应，从而达到了用软件来模拟3D音效的效果。为了模拟出比较真实的3D音效，DirectSound提供了相应的接口对象来实现，但是使用3D声效可能会带来额外的算法开销，如果一定要加入3D声效，则可以利用2D声效的实现方法，简单地模拟出3D效果。

对于3D音效的编程，首先就是要设置一个发声者，然后设置发声者的位置、声音锥的内外锥角、声音锥方向、发声者的声音速度和设置声音可以达到的最大和最小距离。

对于DirectSound的3D声效播放，需要使用DSBCAPS_CTRL3D标志创建一个主缓冲区，由主缓冲区创建一个IDirectSound3DListener8听者对象，调用听者对象提供的函数，设置听者的位置和方向（面向方向和头顶方向）；在3D环境中，声源（发声者）可以用一个IDirectSound3DBuffer接口对象来表示，这就要创建一个添加了DSBCAPS_CTRL3D标志位的次缓冲区，并将WAV声音数据装入，这个接口提供了一些函数，可以用来设置和获得声源的一些属性，然后调用QueryInterface取得一个IDirectSound3DBuffer8接口对象，从而创建一个3D声音缓冲区；接着可以进一步设置发声者的声音锥区域和最大最小距离；最后通过IDirectSound3DListener8接口创建听者对象，并设置听者的位置和方向。程序设计实现的流程如图5-1所示。

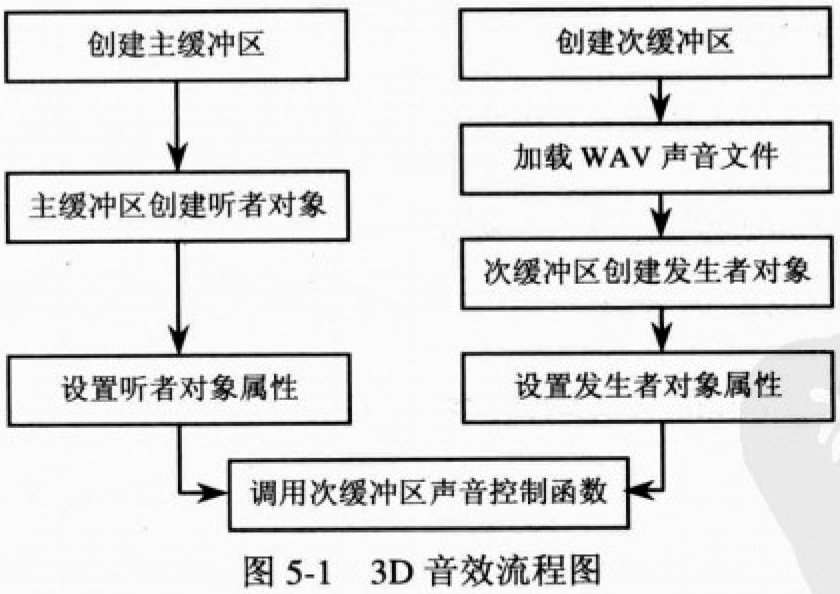


图5-1 3D音效流程图

5.2 DirectSound 3D缓冲区

3D音效与2D音效的不同之处就是加入了空间感，举例来说明，如果在读者的左边有只小狗在叫，那么读者很容易地听出来声音是从读者的左侧传来的；而如果这只小狗边跑边叫

着朝自己所在的方向而来，那么将听到的狗叫声也会越来越大。

为了产生3D的声音效果，DirectSound提供了方法和接口来仿真真实世界中声音的发出和倾听。

3D音效的原理就是根据发声者与倾听者的位置、声音速度和发音锥体区域等信息，计算倾听者的声音强度，以决定播放的声音大小。本质上是根据3D的空间因素，确定倾听者听到的声音的大小。发声者和倾听者这两者存在着一些影响声音效果的因素，因此形成具有空间感的3D音效。下面来介绍几个与3D声音相关的特性，这些特性在DirectSound中都已经定义并且可对这些特性进行设定。而在程序中便可从发声者与倾听者之间的相互关系来定义这些特性，以产生具有变化的3D音效。

读者可以用3D为声音创建一个和3D图像结合的3D世界。在应用程序中可以知道声源和听者在空间中的位置，也可以让移动的声音产生多普勒效应，还可以控制一些如声音随距离衰减的速度等物理参数。DirectSound可以在2个扬声器或多个听筒上创建虚拟3D效果。在WDM（Windows驱动模式）驱动下，如果用户在控制面板里面选择了环场扬声器的话，它可以支持多声道系统。

5.2.1 获取3D缓冲区对象

IDirectSound3DBuffer8接口通常从DirectSound次缓冲区获取，次缓冲区通过DSBUFFERDESC结构的dwFlags成员设置DSBCAPS_CRTL3D标志位创建。

如果不使用DirectMusic创建和管理自己的DirectSound次缓冲区，则应用程序通过调用QueryInterface来取得一个IDirectSound3DBuffer8接口对象。下面的代码对此作了演示。lpDsbSecondary是一个IDirectSoundBuffer8接口。

```
    LPDIRECTSOUND3DBUFFER8 lpDs3dBuffer;
    HRESULT      hr=lpDsbSecondary->QueryInterface(IID_IDirectSound3DBuffer8    ,
(LPVOID*)&lpDs3dBuffer);          //通过QueryInterface获取DirectSoundBuffer8接口
```

下面的代码完整地创建了一个3D音效缓冲区。

```
    WAVEFORMATEX waveFormat;
    ZeroMemory(&waveFormat,sizeof(WAVEFORMATEX));
    waveFormat.wFormatTag=WAVE_FORMAT_PCM;                          //WAV文件格式
    waveFormat.nChannels=pWaveHeader->Channels;                     //声道数
    waveFormat.nSamplesPerSec=pWaveHeader->SampleRate;              //播放频率
    waveFormat.wBitsPerSample=pWaveHeader->BitsPerSample;           //位数
    waveFormat.nBlockAlign=(waveFormat.wBitsPerSample/8)*           //块对齐的大小
                   waveFormat.nChannels;
    waveFormat.nAvgBytesPerSec=waveFormat.nSamplesPerSec *          //每秒采样的平均字节数
                     waveFormat.nBlockAlign;
    //缓冲区结构体
    DSBUFFERDESC bufferDesc;        //DSBUFFERDESC结构体对象
    ZeroMemory(&bufferDesc,sizeof(DSBUFFERDESC));
    bufferDesc.dwSize=sizeof(DSBUFFERDESC);
    bufferDesc.dwFlags=DSBCAPS_CTRL3D|DSBCAPS_CTRLVOLUME|
```

```
    DSBCAPS_GLOBALFOCUS|DSBCAPS_STATIC;            //3D音效控制说明
    bufferDesc.dwBufferBytes=pWaveHeader->DataSize;
    bufferDesc.lpwfxFormat=&waveFormat;
    //先获得IDirectSoundBuffer缓冲区接口对象
    IDirectSoundBuffer* pDirectSoundBuf=NULL;
    If(FAILED(pDirectSound->CreateSoundBuffer(&bufferDesc,&pDirectSoundBuf,NULL)))
      Return false;
    //再获得IDirectSound3DBuffer8缓冲区接口对象（即发声者对象）
    IDirectSound3DBuffer8 *pDirectSound3DBuf8=NULL;
    //DirectSound3DBuffer指针为NULL
    If(FAILED(pDirectSoundBuf->QueryInterface(IID_IDirectSound3DBuffer8,
                (void**)&pDirectSound3DBuf8)))   //获取Direct3DBuffer8接口
    {
      pDirectSoundBuf->Release();                //释放捕捉
      return false;
    }
```

5.2.2 最大距离和最小距离

当听者越来越接近一个声源时，所听到的声音会越来越大，而当听者接近声源的距离减小到一半时，所听到的音量会是原来的 2 倍。但是当这个距离在到达了某一距离时，所听到的音量就不再继续增加了，这段距离就是声源的最小距离。

声源的最小距离，就是声音的音量开始随着距离衰减的起始点。例如，对于一个喷气式飞机，这个最小距离可能是 100m，但是对于一只蜜蜂，这个最小距离可能是 2cm。在这种情况下，当听者距离飞机 200m 时，所听到飞机的音量就要减少一半；但是距离蜜蜂 4cm 时，听到的蜜蜂的声音就要减少一半。图 5-2 表明最大和最小距离对飞机和蜜蜂音量的影响。

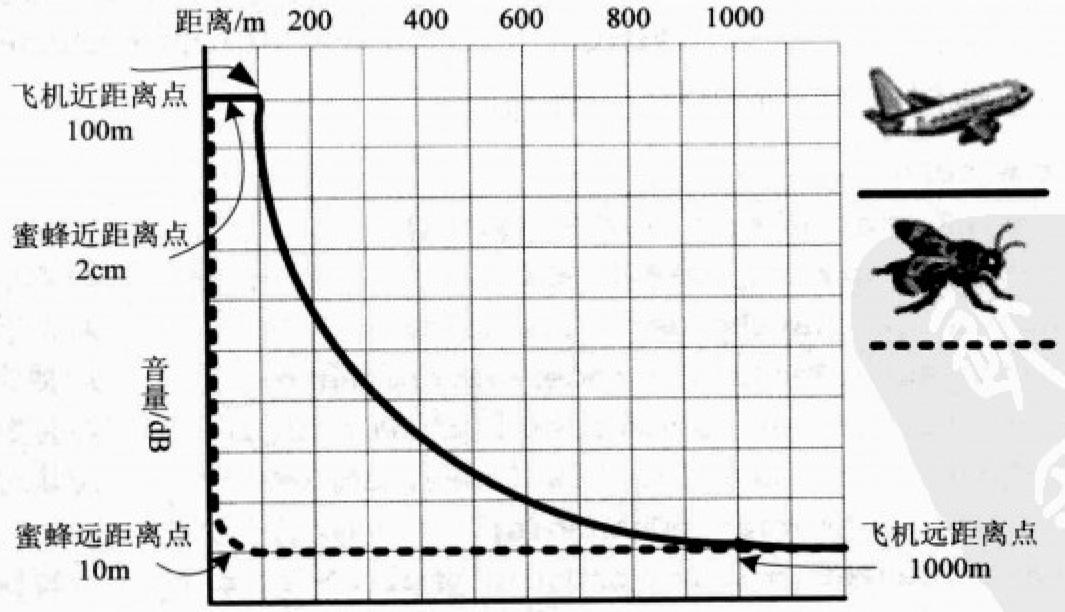


图 5-2 声源的最大距离和最小距离

对于 DirectSound 声音缓冲区来说，默认的最小距离是 DS3D_DEFAULTMINDISTANCE，被定义为 1 个单位。在默认的距离因子的情况下就是 1m。在此默认值下，当声音在 1m 处时，听者听到的音量是完整的音量，在 2m 处所听到的声音是一半的音量，4m 处则是 1/4 音量，

依次类推。对于大多数的声音来说，都可能要设定一个比较大的最小距离，以便当声音运动的时候，声音不会衰减得那么快。

当一个声源离听者的距离越来越远时，所听到的声音会越来越小。当达到某个距离之外时，听者所听到的声音不再减小，这个距离就是最大距离。对于 DirectSound3D 缓冲区默认的最大距离 DS3D_DEFAULTMAXDISTANCE 是 1 000 000 000，就是说当声音远远超过听觉范围之外时，声音的衰减还在继续。在 VXD 驱动模式下，为了避免不必要的处理，当创建缓冲区的时候就应该设定一个合理的最大距离并且还要设定 DSBCAPS_MUTE3DATMAXDISTANCE 标志位。最大距离同时也可以用来避免一个声音听不见。例如，如果将一个声音的最小距离设置为 100m，那么声音可能在 1000m 处就衰减得听不到了。但如果将最大距离设置为 800m，那么无论声音在多远处，都可以保证能听到 1/8 的音量。首先要假设没有设置 DSBCAPS_MUTE3DATMAXDISTANCE 标志位，默认的距离单位是 m。

5.2.3 处理模式

声音缓冲区有 3 种处理模式：正常（Normal）、头部相关（Head-relative）和禁用（Disabled）。

- 在正常模式下，声源的位置和方向在空间里是不变的。这种模式适合声源相对于听者不会移动或者转动的情况。
- 在头部相关的模式下，声源的 3D 特性和听者当前的位置、速度和方法有关。当听者移动或者转动时，3D 缓冲区就会自动在空间中重新调整位置。头部相关模式可以处理如苍蝇围绕听者的头部嗡嗡叫的声音。那些一直随听者移动的声音根本不是 3D 声音。
- 在禁用的模式下，3D 声音的处理是无效的，声音似乎是从听者的同步中心产生的。

5.2.4 声音圆锥

对于 3D 音效的编程，首先要设定一个发声者，然后设置发声者的位置、声音锥的内外锥角、声音锥方向、发声者的声音速度和声音可以到达的最小最大距离。

一个没有方向的声音在一定的距离下，在各个方向上的振幅是相同的。有方向的声音在该方向上的振幅最大，其他方向上弱。声音圆锥是描述有方向的声音响度的模型，声音圆锥由内锥和外锥构成。外锥的角度必须大于或等于内锥的角度。

在圆锥的内部，当听者相对于声源的角度小于内锥角度时，声音的音量就和没有声音圆锥是一样的。主要把音量、听者和声源的距离、听者方向等参数代入计算就可以了。

当角度大于或等于外锥角度时，音量就会在正常音量值的基础上乘以一个指定的小于 1 的因子来进行计算。外锥音量用百分之一分贝为单位来表示，通常它是一个负值，因为它表示的是音量的衰减值。

在内锥和外锥之间是一个过渡带，从内部音量到外部音量，这个音量会随着角度的增加而减小，图 5-3 显示了声音的圆锥效应。

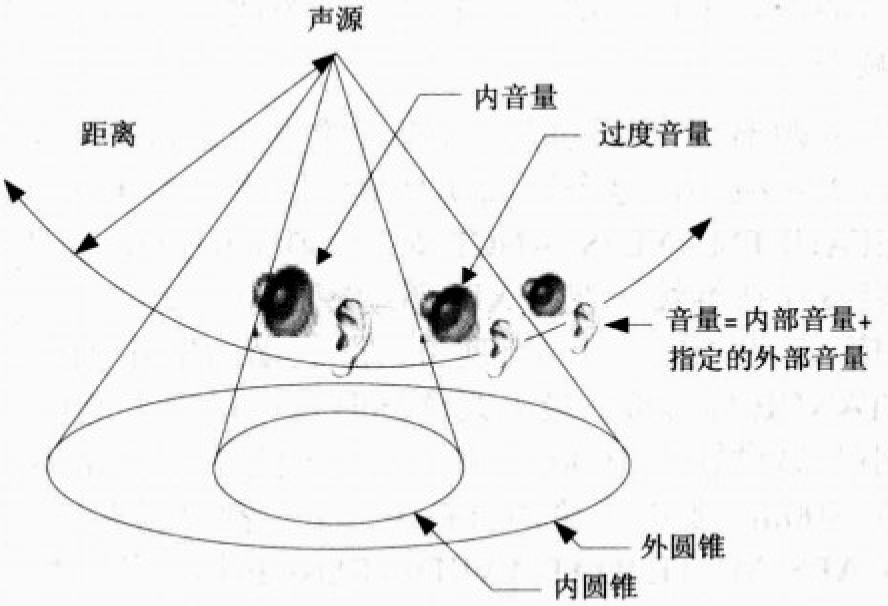


图 5-3　声音圆锥图

而每一个 3D 缓冲区都会有一个声音圆锥，但是在默认的情况下，声源是各方向都均等的，也就是在外锥的声音不会衰减，内锥和外锥的角度都是 360 度。除非应用程序改变了这些值，否则声音就不会有方向感。

适当地利用 3D 声音可以给应用程序添加比较奇特的效果。例如，可以将一个声源放在房间的中心，设定它的方向朝向房门的方向，然后将声音的锥体角度刚好包含门的宽度，外锥角度稍微宽一些，而且设定外锥音量为 0，即无声。那么听者只有在靠近门的时候才能听到声音，在正对门的位置听到的声音最大。

5.3　DirectSound 3D虚拟听众

上一节详细介绍了 3D 音效的一些基本概念，包括缓冲区的创建、最大最小距离及处理模式等。对于 3D 音效编程，重要的一点就是要首先建立发生者（即声源），这在上一节也已介绍过。但同时另外重要的一点就是要创建听者（听众），本节随后将详细介绍听者的概念，以及影响 3D 音效的因素。

5.3.1　获得3D听众

为了获得 3D 听众，必须通过主缓冲区来获得 IDirectSound3DListen8 接口，应用程序应该创建一个主缓冲区对象，然后获得 Listener 指针，通过 IDirectSound::CreateSoundBurrer 创建一个主缓冲区。并在主缓冲区中设定 DSBCAPS_CTRL3D 标志位，使其具有 3D 音效的控制功能。下面是建立 3D 听者的程序片段。

```
GetListener(LPDIRECTSOUND8 lpds, LPDIRECTSOUND3DLISTENER8* ppListener)
{
DSBUFFERDESC dsbd;                              //创建结构体对象
LPDIRECTSOUNDBUFFER lpdsbPrimary;               //建立主缓冲区指针
LPDIRECTSOUND3DLISTENER8 lp3DListener = NULL;   //3D听者指针
HRESULT hr;
```

```
    ZeroMemory(&dsbd, sizeof(DSBUFFERDESC));
    dsbd.dwSize = sizeof(DSBUFFERDESC);        //结构体大小
    dsbd.dwFlags = DSBCAPS_CTRL3D | DSBCAPS_PRIMARYBUFFER;
                                               //添加3D声音控制说明
    if (SUCCEEDED(hr = lpds->CreateSoundBuffer(&dsbd, &lpdsbPrimary, NULL)))
    {                                          //创建声音缓冲区
     hr = lpdsbPrimary->QueryInterface(IID_IDirectSound3DListener8,
     (LPVOID *)ppListener);                    //获取DirectSound3DListener8接口
     lpdsbPrimary->Release();                  //释放缓冲区
    }
    return hr;
    }
```

如此便建立了一个3D听者对象，接下来便可使用LPDIRECTSOUND3DLISTENER接口下的各种方法来设定听者的特性，常用的方法如表5-1所示。

表5-1 LPDIRECTSOUND3DLISTENER接口下的各种方法

方法	说明
SetPosition	设定听者的位置
GetPosition	获取听者的位置
SetVelocity	设定听者的速度
GetVelocity	获取听者的速度
SetOrientation	设定听者面对的方向
GetOrientation	获取听者面对的方向
SetAllParameters	设定听者的全部特性
GetAllParameters	获取听者的全部特性

5.3.2 听众的空间参数

和真实世界一样，在一个虚拟3D环境中，声音是以点接收的方式存在的，只和听到声音的位置有关。在Microsoft DirectSound的应用程序中的3D特效不仅和声源的位置、方向、速度有关，并且也和收听者的位置、方向和速度有关。

在默认情况下，听者固定在各个方向向量的零点，鼻子朝向z轴方向，头顶部朝向y轴方向。通过获取一个听者的对象，应用程序可以改变虚拟世界中听者的位置和运动，听者对象还控制着常用的声学环境中的参数，比如多普勒变换的数量和音量随距离衰减的速度。

在3D世界中，像声源一样听者也有自己的位置、速度和方向。听者的方向由听者头部的两个方向向量决定，如图5-4所示，向上的向量指向头的顶部，向右的向量指向听者面部的正前方。在默认情况下，向前的向量是（0.0，0.0，1.0），朝向头的顶部的向量是（0.0，1.0，0.0），如果需要，DirectSound可以调整向前的向量，因为它在顶部向量的右向。

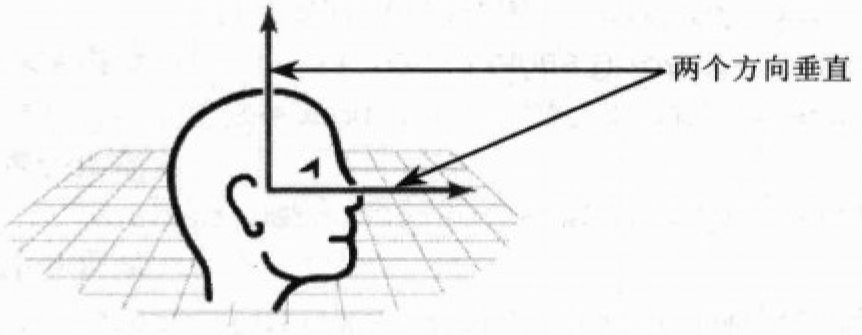


图 5-4　听者方向示意图

当一个声源移动时，应用程序就应指定它的位置和速度。

位置用 3 个坐标轴上的分量来描述，单位是设定的距离单位，它可以以空间和听者为参照，这依赖于处理模式。

速度用 3 个坐标轴上的速度分量来描述，单位是 m/s。在默认情况下，距离单位是 m，速度单位是 m/s。速度一般被 DirectSound 用来计算声音的多普勒效应。

通过下面 6 个函数可以获得听众的空间参数：位置和速度。

IDirectSound3DListener8::SetPosition 方法用来设置三维空间中听众的位置。

此函数原型为：

```
HRESULT SetPosition(
  D3DVALUE x,          //x方向
  D3DVALUE y,          //y方向
  D3DVALUE z,          //z方向
  DWORD dwApply        //
);
```

x、y、z 的值与听众的方向向量相关。DirectSound 可以适当调整这些值，防止溢出，参数 dwApply 常用值如表 5-2 所示。

表 5-2　dwApply 常用值

值	描述
DS3D_DEFFRRED	此值被设置必须是应用程序调用 IDirectSound3DListener8::CommitDeferredSettings 方法，它允许程序改变几个设置
DS3D_IMMEDIATE	设置后被立即应用，能引起系统为所有 3D 缓冲区重新计算匹配的值

IDirectSound3DListener8::GetPosition 方法来获取空间听众的位置；此方法的调用和 SetPosition 方法类似。

IDirectSound3DListener8::SetVelocity 方法用来设置三维空间中听众的速度。

此函数的原型为：

```
HRESULT SetVelocity(
  D3DVALUE x,     //x方向
  D3DVALUE y,     //y方向
  D3DVALUE z,     //z方向
  DWORD dwApply
);
```

x、y、z的值与听众的方向向量相关。DirectSound可以适当调整这些值，防止溢出，参数dwApply常用值如表5-3所示。

IDirectSound3DListener8::GetVelocity方法来获取空间听众的速度；此方法调用和SetPosition类似。IDirectSound3DListener8::SetOritation方法用来设置三维空间中听众的运动方向。

```
HRESULT SetOrientation(
  D3DVALUE xFront,  //参数意义见下
  D3DVALUE yFront,
  D3DVALUE zFront,
  D3DVALUE xTop,
  D3DVALUE yTop,
  D3DVALUE zTop,
  DWORD dwApply
  );
```

参数xFront, yFront, zFront的值与听者前方方向向量的值相关。参数xTop, yTop, zTop的值与听者顶部方向向量的值相关，参数dwApply常用值如表5-3所示。

表5-3　dwApply的常用值

值	描述
DS3D_DEFERRED	设置时必须是应用程序调用IDirectSound3DListener8::CommitDeferredSettings方法，它允许程序改变几个设置
DS3D_IMMEDIATE	设置后被立即应用，能引起系统为所有3D缓冲区重新计算匹配的值

IDirectSound3DListener8::GetOrientation方法来获取空间听众的运动方向。此方法调用和SetOrientation类似。

5.3.3　距离因子与Doppler效应

在DirectSound 3D虚拟听众中，距离因子和Doppler效应是很重要的概念。距离因子通过改变速度来影响3D听众。同时可以通过改变Dippler效应来增强和减弱3D音效。

1．距离因子

距离因子就是向量单位中的m。默认的距离因子是1.0。如果一个缓存的速度是（2.0，0.0，0.0），则声源就被认为是沿着x轴正向移动2m/s。使用不同的刻度单位，3D的应用程序就要相应地改变距离因子。

假设应用程序中的基本刻度单位是英尺或者0.3048m。将3D听者的距离因子属性设置为0.3048。则位置和速度的属性就将以英尺来度量。

距离因子主要通过改变实际速度来影响多普勒转换，该速度用n单位/秒表示。它不直接影响高低频规律性衰减，因为随着距离衰减，速度是基于未指明单位的最小距离的。如果为一个给定的声音，在2个单位处设置了最小距离，则音量将在4个单位距离处减半，无论那

些声音是以英尺、米还是以其他任何方法来度量的。

通过调用 IDirectSound3DListener8::SetDistanceFactor 来设置距离因子。SetDistanceFactor 方法来设置距离因子，它以 m 为单位。此方法的原型是：

```
HRESULT SetDistanceFactor(
  D3DVALUE flDistanceFactor,        //定义距离因子
  DWORD dwApply
);
```

其中，flDistanceFactor 是距离因子。D3DVALUE 被定义为浮点型。dwApply，当设置被应用时，这个值才被指定。这个值是表 5-3 的值其中之一。

如果此方法成功，则返回值是 DS_OK。如果失败，则返回值是 DSERR_INVALIDPARAM。

距离因子范围从 DS3D_MINDISTANCEFACTOR 到 DS3D_MAXDISTANCEFACTOR，在 Dsound.h 中它们分别被定义 FLT_MIN 和 FLT_MAX，默认的值是 DS3D_DEFAULTDISTANCEFACTOR 或者 1.0。

通过调用 IDirectSound3DListener8::GetDistanceFactor 来获得距离因子。

GetDistanceFactor 方法获取距离因子，以 m 为单位。此方法原型为：

```
HRESULT GetDistanceFactor(
  D3DVALUE * pflDistanceFactor    //获取距离因子指针
);
```

pflDistanceFactor 是一个指针，用来获得距离因子值。D3DVALUE 被定义为浮点型。如果此方法成功，则返回值是 DS_OK。如果失败，则返回值是 DSERR_INVALIDPARAM。

2. Doppler 效应

Doppler 效应，当任何的缓冲区或者听者以一个速度移动时，DirectSound 会自动地创建多普勒变换效应。这个效应需要不断累积，如果听者和声音缓冲区都在不时地移动，则系统会自动地计算出相关的速度并调整多普勒效应。

为了在应用程序中有现实的多普勒变换效应，就必须计算出任何正在移动的对象的速度，并且为声源和听者设置适当的速度。在一个特殊的例子中用户可以随意夸大或减小这个值，为的就是创建特殊的效应。也可以通过改变多普勒因子来全部地增加或减少多普勒转换效应。

多普勒因子范围从 DS3D_MINDOPPLERFACTOR 到 DS3D_MAXDOPPLERFACTOR，被分别地定义为 0.0 和 10.0。0 值意味着没有多普勒效应被应用到一个声音。其他的每一个值代表真实世界中的各种多普勒效应。

DirectSound 中可以用 IDirectSound3DListener::SetDopplerFactor 方法来设置多普勒因子参数。此函数原型为：

```
HRESULT SetDopplerFactor(
  D3DVALUE flDopplerFactor,  //设置多普勒因子
  DWORD dwApply
);
```

flDopplerFactor 是多普勒因子，D3DVALUE 被定义为浮点型，当设置被应用时，dwApply

才被指定。此值必是表5-3中的其中之一。

如果此方法成功，则返回值是DS_OK。如果失败，则返回值是DSERR_INVALIDPARAM。

DirectSound中可以调用IDirectSound3DListener8::GetDopplerFactor来获取多普勒效果。此函数的原型为：

```
HRESULT GetDopplerFactor(
  D3DVALUE * pflDopplerFactor  //获取多普勒因子
);
```

pflDopplerFactor是一个指针，来获取多普勒因子值。D3DVALUE被定义为浮点型。如果此方法成功，则返回值是DS_OK。如果失败，则返回值是DSERR_INVALIDPARAM。

而DirectSound可以根据听众和声源的相对位置加上多普勒效应。在程序中，多普勒效果可以忽略不计，也可以与现实世界相同，取决于多普勒因子的设定。多普勒因子为0，表示不应用多普勒效果；取值为1，表示多普勒效果与现实世界一致；取值为2时，表示多普勒效果为现实世界的2倍，以此类推。

5.3.4 滚降因子与延迟设置

在DirectSound的3D音效中，有两个很重要的参数，那就是滚降和延迟。当听众与声源的距离比较远时，则声音必然要衰减，声音的衰减即为滚降。当对3D声音和听众进行参数设置时，必然花费时间，结果会影响混音的效果，因此可以通过延迟设置来解决。下面介绍了滚降和延迟的概念和设置。

1. 滚降因子

滚降被应用到声音时就等于衰减。它基于远离声源的听众。DirectSound能忽略滚降，能夸大滚降，或者在真实世界中给它相同的效果，它依赖于一个叫做滚降因子的全局参数。

滚降因子的范围从DS3D_MINROLLOFFFACTOR到DS3D_MAXROLLOFFFACTOR，被分别定义为0.0和10.0。DS3D_MINROLLOFFFACTOR意味着尽管远离听者，但声音仍然是完整音量。其中的每个值代表真实世界的多个滚降。滚降因子是全局的。为了改变单个声音缓冲区的距离影响，必须为缓冲区设置最小距离。DirectSound中可以通过调用IDirectSound3DListener8::SetRolloffFactor方法来设置滚降因子。SetRolloffFactor方法设置滚降因子，它决定衰减距离的速率。此函数的原型是：

```
HRESULT SetRolloffFactor(
  D3DVALUE flRolloffFactor,  //定义滚降因子
  DWORD dwApply
);
```

其中，flRolloffFactor是衰减因子，D3DVALUE被定义为浮点型，当设置被应用时，dwApply值才被指定。这个值是表5-3的值的其中之一。

如果此方法成功，则返回值是DS_OK。如果失败，则返回值是DSERR_INVALIDPARAM。

2．延迟设置

对 3D 声音缓存和听者的每个设置的改变，将会导致 CPU 花费周期进行混音。要使改变 3D 设置而产生的性能影响降至最低，可以在 3D 听者对象和所有 3D 缓冲区对象上设置延迟属性。当该属性设置为 True 时，调用 3D 对象的 IDirectSound3D Listener8::CommitDeferred Settings 不会有其他属性发生变化。

注意：

> 延迟设置通过立即设置而被重写。比如，如果将延迟设置为 True 并将听者速度设置为（1.0，0.0，0.0），然后将延迟设置为 False 并将速度设置为（2.0，0.0，0.0），速度立即变为（2.0，0.0，0.0），当 CommitDeferredSettings 调用时将不再改变。

DirectSound 中可以通过调用 IDirectSound3DListener8::CommitDeferredSettings 方法来延迟设置，当最后调用此方法时，CommitDeferredSettings 方法将做任意延迟设置。函数原型是：

```
HRESULT CommitDeferredSettings();
```

此函数没有参数。如果此方法成功，则返回值是 DS_OK。如果失败，则返回值是 DSERR_INVALIDPARAM。

5.4 系统效果演示

上面介绍了 DirectSound 3D 缓冲区和 DirectSound 3D 虚拟听众的部分基本实现原理，下面就来介绍本案例语音特效器的具体实现过程及其系统效果的演示。那么怎么产生 3D 音效呢？一般来说，要产生 3D 音效，主要有下面的几种情况。

- 声源不动，听者在模拟的 3D 空间进行运动。
- 听者不动，让声源在模拟的 3D 空间进行运动。
- 声音和听者同时运动。

而在本案例中就是用的第二种情况，即听者不动，而让声源在 3D 控件中进行运动，来实现 DirectSound 中的 3D 特效。

5.4.1 具体的实现步骤

STEP 01 新建一个基于对话框的 MFC 项目，将其命名为 DirectSound 3DSound，然后把所需的链接文件加入到链接库中，再在对话框上放置按照表中相应的控件，如表 5-4 所示。

表 5-4 控件属性

ID	标题	控件类型	说明
ID_BUTTON1	打开文件	Button Control	打开声音文件
ID_PALY	播放	Button Control	播放声音文件

（续表）

ID	标题	控件类型	说明
ID_STOP	停止	Button Control	停止声音文件
ID_OK	退出	Button Control	退出播放
IDC_EDIT1	无	Edit Control	显示文件路径
IDC_EDIT2	无	Edit Control	显示文件状态
IDC_STATIC	状态	Text Control	显示状态
IDC_STATIC	最大距离	Text Control	显示最大距离
IDC_STATIC	最小距离	Text Control	显示最小距离
IDC_STATIC	多普勒因子	Text Control	显示多普勒因子
IDC_STATIC	Rolloff 因子	Text Control	显示滚降因子
IDC_MINDISTANCE	0	Text Control	显示最小距离值
IDC_MAXDISTANCE	0	Text Control	显示最小距离值
IDC_DOPPLERFACTOR	0	Text Control	显示多普勒因子值
IDC_ROLLOFFFACTOR	0	Text Control	显示滚降因子值
IDC_MAXDISTANCE_SLIDER	无	Slider Control	滑动条
IDC_MINDISTANCE_SLIDER	无	Slider Control	滑动条
IDC_DOPPLER_SLIDER	无	Slider Control	滑动条
IDC_ROLLOFF_SLIDER	无	Slider Control	滑动条
IDC_HORIZONTAL_SLIDER	无	Slider Control	声源水平运动
IDC_VERTICAL_SLIDER	无	Slider Control	声源垂直运动
IDC_RENDER_WINDOW	无	Picture Control	显示声源运动
IDC_STATIC	声音属性	Group Control	无
IDC_STATIC	声源运动	Group Control	无

STEP 02 在对话框的“Caption”属性中输入“DirectSound 3DSound”，如图 5-5 所示。ID 设为“IDD_DSOUND3DPLAYDEMO_DIALOG”，如图 5-6 所示。

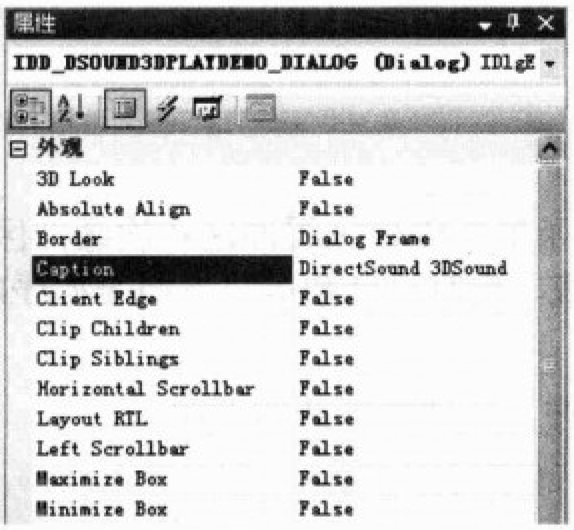


图 5-5 对话框标题为“DirectSound 3DSound”

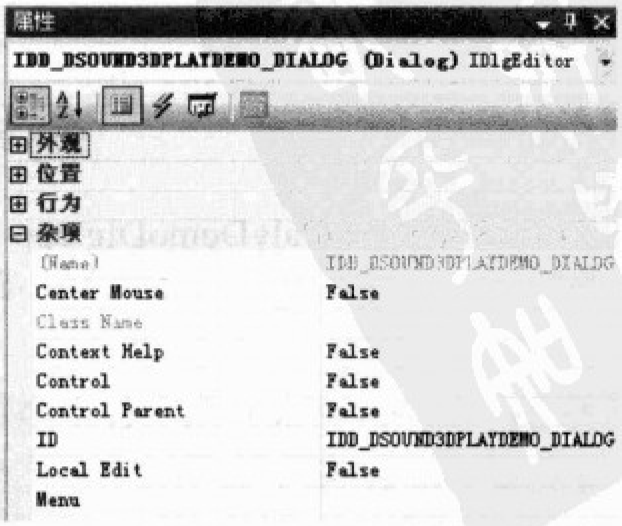


图 5-6 设置对话框 ID

STEP 03 对于选择 3D 算法对话框，在此对话框的“Caption”属性中输入“IDD_3D_ALGORITHM”，然后在对话框中添加表中相应的属性，控件属性如表 5-5 所示。添加过程与上图类似。

表 5-5　控件属性

ID	标题	控件属性
IDC_NO_VIRT_RADIO	&No Virtualization (WDM or VxD. CPU efficient, but basic 3-D effect)	Radio Button Control
IDC_HIGH_VIRT_RADIO	&High Quality (WDM only. Highest quality 3D audio effect, but uses more CPU)	Radio Button Control
IDC_LIGHT_VIRT_RADIO	&Light Quality (WDM only. Good 3-D audio effect, but uses less CPU than High Quality)	Radio Button Control

5.4.2　程序的实现代码

上面详细介绍了 3D 音效的一些重要概念，包括 DirectSound 3D 缓冲区及 DirectSound 3D 虚拟听众等，这些都是 3D 音效编程中的重点。为了达到本案例的设计目标，根据上述这些重要思想，下面设计具体的实现代码。首先给出了程序设计的流程图，帮助读者总体把握，如图 5-7 所示。

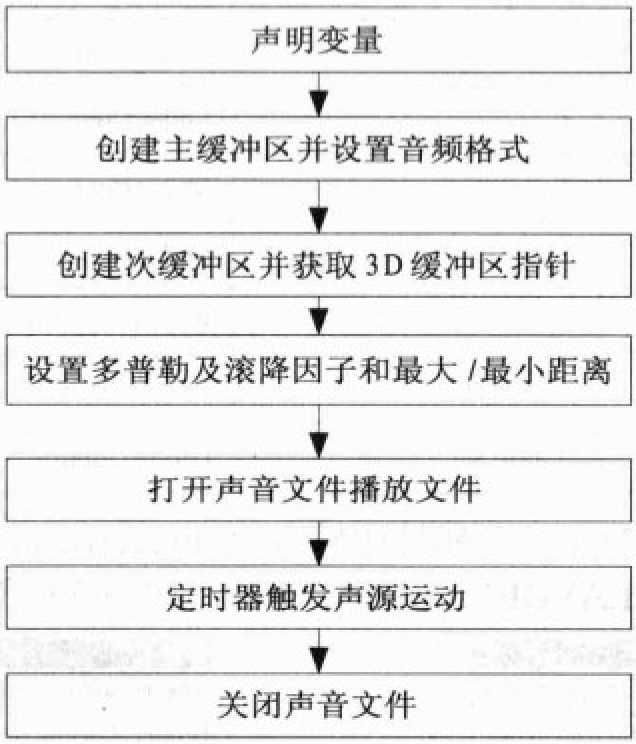


图 5-7　实现代码流程图

在 DirectSound 3DPalyDemoDlg.cpp 文件中包含的函数有创建主缓冲区和次缓冲区函数、设置多普勒因子、滚降因子和最大/最小距离、播放函数、定时器函数等。下面是程序的实现代码。

STEP 01 在 WaveFile.h 头文件中添加下面的头文件，方便对函数的调用。

```
#include <mmsystem.h>
#include <dsound.h>
```

STEP 02 在 DirectSound 3DPalyDemoDlg.cpp 文件中进行变量的定义。

```
LPDIRECTSOUND8        g_pDsd = NULL;                    //Directsound 对象指针
LPDIRECTSOUNDBUFFER   g_pDSBuffer= NULL;                //辅助缓冲区指针
LPDIRECTSOUND3DBUFFER g_pDS3DBuffer= NULL;              //3D 声源对象指针
LPDIRECTSOUND3DLISTENER  g_pDSListener = NULL;          //3D 听者指针
DS3DBUFFER          g_dsBufferParams;                   //3D 缓冲区属性
DS3DLISTENER         g_dsListenerParams;                //听者属性
CWaveFile             *g_pWaveFile= NULL;               //读写wave文件用到的一个类对象
BOOL                g_bPlaying = FALSE;                 //是否正在播放
LPDIRECTSOUNDNOTIFY8  g_pDSNotify = NULL;               //通知点指针
DSBPOSITIONNOTIFY     g_aPosNotify[MAX_AUDIO_BUF];//设置通知标志的数组
DWORD g_dwNextWriteOffset = 0;                          //下一个写偏移量
```

在这里定义这些变量，主要就是方面下便程序设计的直接进行。

STEP 03 初始化 Directsound 的主缓冲区，并设置主缓冲区格式。

```
//初始化DirectSound
HRESULT hr;
if(FAILED(hr = DirectSoundCreate8(NULL,&g_pDsd,NULL)))
    return FALSE;
if(FAILED(hr = g_pDsd->SetCooperativeLevel(m_hWnd,DSSCL_PRIORITY)))
    return FALSE;                                 //设置协调级别
LPDIRECTSOUNDBUFFER pDSBPrimary = NULL;
DSBUFFERDESC   dsbdesc ;                          //设置结构体对象
ZeroMemory(&dsbdesc,sizeof(DSBUFFERDESC));
     dsbdesc.dwSize = sizeof(DSBUFFERDESC);
     dsbdesc.dwFlags = DSBCAPS_CTRL3D | DSBCAPS_PRIMARYBUFFER;
                                                  //表示3D控制
if(FAILED(hr = g_pDsd->CreateSoundBuffer(&dsbdesc,&pDSBPrimary ,NULL)))
    return FALSE;                                 //创建主缓冲区
if(FAILED(hr=pDSBPrimary->QueryInterface(IID_IDirectSound3DListener,
(VOID**)&g_pDSListener ) ) )                      //获取3D听者接口
   return FALSE;
WAVEFORMATEX wfx;                                 //下面设置缓冲区的音频格式
ZeroMemory( &wfx, sizeof(WAVEFORMATEX) );
wfx.wFormatTag = (WORD) WAVE_FORMAT_PCM;          //格式类型
wfx.nChannels = WAVECHANNEL   ;                   //通道数
wfx.nSamplesPerSec = WAVESAMPLEPERSEC  ;          //采样率
wfx.wBitsPerSample = WAVEBITSPERSAMPLE ;          //位数
wfx.nBlockAlign = (WORD) (wfx.wBitsPerSample / 8 * wfx.nChannels); //块对齐大小
wfx.nAvgBytesPerSec = (DWORD) (wfx.nSamplesPerSec * wfx.nBlockAlign);
```

此段代码主要实现初始化 DirectSound 并创建主缓冲区，设置缓冲区的一些参数和音频格式，通过主缓冲区获取听者对象接口。

STEP 04 选择文件。

```
void CDirectSound 3DSoundDlg::OnSelectFile()
```

```
{
GUID   guid3DAlgorithm = GUID_NULL;
int      nResult;
HRESULT hr;
static char strFileName[MAX_PATH] ="";    //声明一个字符数组，用做文件名
static char strPath[MAX_PATH] = "";
OPENFILENAME ofn = { sizeof(OPENFILENAME), m_hWnd, NULL,
//用OPENFILENAME结构声明一个对象
    TEXT("Wave Files\0*.wav\0All Files\0*.*\0\0"), NULL,
    0, 1, strFileName, MAX_PATH, NULL, 0, strPath,
    TEXT("Open Sound File"),
    OFN_FILEMUSTEXIST|OFN_HIDEREADONLY, 0, 0, //设置对话框风格
    TEXT(".wav"), 0, NULL, NULL };
if(strPath[0] == '\0')
{
    if( strcmp( &strPath[strlen(strPath)], TEXT("\\") ) )
        strcat( strPath, TEXT("\\") );
    strcat( strPath, TEXT("MEDIA") );
}
if(GetOpenFileName(&ofn) != TRUE)          //向ofn保持必要的参数
    return ;
}
```

此段代码实现了选择要播放声音文件的功能。同时也设置了对话框的显示风格。

STEP 05 创建次缓冲区。

```
DSBUFFERDESC dsbd;                                  //定义结构体对象
ZeroMemory( &dsbd, sizeof(DSBUFFERDESC) );
dsbd.dwSize = sizeof(DSBUFFERDESC);         //获取结构体大小
dsbd.dwFlags=DSBCAPS_CTRL3D|DSBCAPS_GLOBALFOCUS|   DSBCAPS_CTRLPOSITIONNOTIFY
|DSBCAPS_GETCURRENTPOSITION2;
                                                    //缓冲区的用途标志，
dsbd.dwBufferBytes =g_pWaveFile->GetSize();
//如果是静态缓冲区，则缓冲区大小就是文件大小
dsbd.guid3DAlgorithm = guid3DAlgorithm;
dsbd.lpwfxFormat= g_pWaveFile->m_pwfx;
if(FAILED(hr = g_pDsd->CreateSoundBuffer(&dsbd,&g_pDSBuffer,NULL)))
    return ;                                        //创建缓冲区
if(FAILED(hr=g_pDSBuffer->QueryInterface(IID_IDirectSound3DBuffer,
(VOID**)&g_pDS3DBuffer )))                          //获取DirectSound3DBuffer接口
    return ;
   g_dsBufferParams.dwSize = sizeof(DS3DBUFFER);
g_pDS3DBuffer->GetAllParameters( &g_dsBufferParams ); //设置所有参数
//设3D缓冲区参数
g_dsBufferParams.dwMode = DS3DMODE_HEADRELATIVE;
g_pDS3DBuffer->SetAllParameters( &g_dsBufferParams, DS3D_IMMEDIATE );
   if(FAILED(hr = g_pDSBuffer->QueryInterface(IID_IDirectSoundNotify,(LPVOID *)
```

```
&g_pDSNotify ))) //获取通知接口
        return ;
    g_pDSNotify->SetNotificationPositions(MAX_AUDIO_BUF,g_aPosNotify);
    g_pDSNotify->Release();                              //设置通知事件位置
```

此段代码实现创建辅助缓冲区，并通过辅助缓冲区获取 3D 缓冲区指针。

STEP 06 设置多普勒因子、滚降因子、最大距离和最小距离等参数。

```
    FLOAT fDopplerFactor;     //定义多普勒因子
    FLOAT fRolloffFactor;     //定义滚降因子
    FLOAT fMinDistance;       //定义最小距离
    FLOAT fMaxDistance;       //定义最大距离
    CSliderCtrl *pDopplerSlider = (CSliderCtrl*)GetDlgItem(IDC_DOPPLER_SLIDER);
    fDopplerFactor = pDopplerSlider->GetPos();          //获取参数
    CSliderCtrl *pRolloffSlider= (CSliderCtrl*)GetDlgItem(IDC_ROLLOFF_SLIDER);
    fRolloffFactor = pRolloffSlider->GetPos();          //获取参数
    CSliderCtrl*pMinDistSlider= liderCtrl*)GetDlgItem(IDC_MINDISTANCE_SLIDER );
    fMinDistance = pMinDistSlider->GetPos();            //获取参数
    CSliderCtrl*pMaxDistSlider=
    (CSliderCtrl*)GetDlgItem(IDC_MAXDISTANCE_SLIDER );
    fMaxDistance = pMaxDistSlider->GetPos();            //获取参数
    g_dsListenerParams.flDopplerFactor = fDopplerFactor;
    g_dsListenerParams.flRolloffFactor = fRolloffFactor;
    if( g_pDSListener )
    {                                                   //设置所有参数
        g_pDSListener->SetAllParameters( &g_dsListenerParams, DS3D_DEFERRED );
        g_pDSListener->CommitDeferredSettings();        //获取延迟设置
    }
    g_dsBufferParams.flMinDistance = fMinDistance;
    g_dsBufferParams.flMaxDistance = fMaxDistance;
    if( g_pDS3DBuffer )                                 //设置所有参数
        g_pDS3DBuffer->SetAllParameters( &g_dsBufferParams, DS3D_DEFERRED );
    GetDlgItem( IDC_PLAY )->EnableWindow( TRUE );
    GetDlgItem( IDC_STOP )->EnableWindow( FALSE );
    }
```

此段代码主要就是设置多普勒因子及最大/最小距离。并且获取所设置的数值。

STEP 07 处理缓冲区。

```
    void ProcessBuffer()
    {
    DWORD   dwBytesWrittenToBuffer = 0;
    VOID*   pDSLockedBuffer = NULL;
    VOID*   pDSLockedBuffer2 = NULL;
    DWORD   dwDSLockedBufferSize;      //锁存缓冲区1大小
    DWORD   dwDSLockedBufferSize2;     //锁存缓冲区2大小
    HRESULT hr;
       hr= g_pDSBuffer->Lock(g_dwNextWriteOffset,BUFFERNOTIFYSIZE, //锁存缓冲区
```

```
&pDSLockedBuffer,&dwDSLockedBufferSize,
            &pDSLockedBuffer2,&dwDSLockedBufferSize2,0);
if(hr == DSERR_BUFFERLOST)
{
g_pDSBuffer->Restore();        //恢复
g_pDSBuffer->Lock(g_dwNextWriteOffset,BUFFERNOTIFYSIZE,            //锁存缓冲区
&pDSLockedBuffer,&dwDSLockedBufferSize,
         &pDSLockedBuffer2,&dwDSLockedBufferSize2,0);
}
if(SUCCEEDED(hr))
{g_pWaveFile->Read((BYTE*)pDSLockedBuffer,dwDSLockedBufferSize,   //加载数据
&dwBytesWrittenToBuffer);
g_dwNextWriteOffset += dwBytesWrittenToBuffer;
if (NULL != pDSLockedBuffer2)
{ g_pWaveFile->Read((BYTE*)pDSLockedBuffer2,dwDSLockedBufferSize2,
&dwBytesWrittenToBuffer);
g_dwNextWriteOffset += dwBytesWrittenToBuffer;
}
    g_dwNextWriteOffset %= (BUFFERNOTIFYSIZE * MAX_AUDIO_BUF);
    if(dwBytesWrittenToBuffer <BUFFERNOTIFYSIZE )
    {
        FillMemory( (BYTE*) pDSLockedBuffer + dwBytesWrittenToBuffer,
               BUFFERNOTIFYSIZE - dwBytesWrittenToBuffer,
               (BYTE)(g_pWaveFile->m_pwfx->wBitsPerSample == 8 ? 128 : 0 ) );
       g_bPlaying = FALSE;
    }                               //解锁
    hr = g_pDSBuffer->Unlock(pDSLockedBuffer,dwDSLockedBufferSize,
                     pDSLockedBuffer2,dwDSLockedBufferSize2);
}
}
```

此段代码实现了对缓冲区的操作。首先锁存缓冲区，向缓冲区加载声音数据，最后对缓冲区进行解锁。

STEP 08 开始播放。

```
void CDirectSound 3DSoundDlg::OnBnClickedPlay()
{
DWORD  res;
   LPVOID  lplockbuf;    //块缓冲区指针
DWORD  len;              //文件长度
DWORD  dwWrite;
g_pDSBuffer->Lock(0,0,&lplockbuf,&len,NULL,NULL,       //锁存
DSBLOCK_ENTIREBUFFER);
g_pWaveFile->Read((BYTE*)lplockbuf,len,&dwWrite);      //加载
g_pDSBuffer->Unlock(lplockbuf,len,NULL,0);             //解锁
g_pDSBuffer->SetCurrentPosition(0);                    //设置当前位置
g_pDSBuffer->Play(0,0,DSBPLAY_LOOPING);
}
```

此段代码实现播放声音的功能。首先锁存缓冲区，然后向缓冲区加载数据，设置要播放的位置，播放声音。

STEP 09 停止播放。

```
void CDirectSound 3DSoundDlg::OnBnClickedStop()
{
   g_bPlaying =FALSE;
Sleep(500);                    //延迟0.5秒
g_pDSBuffer->Stop();           //调用停止函数
GetDlgItem( IDC_PLAY )->EnableWindow( TRUE); //使能窗口
GetDlgItem( IDC_STOP )->EnableWindow( FALSE );
}
```

此段代码实现播放的停止。置标志位，延迟0.5秒调用停止播放函数实现停止播放功能。

STEP 10 退出。

```
void CDirectSound 3DSoundDlg::OnBnClickedOk()
{
SafeRealse();
if(g_pDSListener)
    g_pDSListener->Release();     //释放听者指针
if(g_pDsd)
    g_pDsd->Release();            //释放捕获
OnOK();
}
```

此段代码实现退出功能。首先安全释放缓冲区，释放听者指针，释放捕获，成功退出。

STEP 11 定时器处理。

触发声源运动，把定时器定义在DirectSound 3DPlayDemoDlg.cpp文件中。

```
void CDirectSound 3DSoundDlg::OnTimer(UINT nIDEvent )
{
FLOAT fXScale;        //X范围
FLOAT fYScale;        //Y范围
fXScale= SliderCtrl*)GetDlgItem(IDC_HORIZONTAL_SLIDER ))->GetPos())/100.0f;
fYScale= (((CSliderCtrl*)GetDlgItem(IDC_VERTICAL_SLIDER ))->GetPos())/100.0f;
FLOAT t = timeGetTime()/1000.0f;
//围绕听者移动声源，最大移动半径为 27.5个单位
D3DVECTOR vPosition;      //位置向量，x，y，z分量
vPosition.x = ORBIT_MAX_RADIUS * fXScale * (FLOAT)sin(t);
vPosition.y = 0.0f;
vPosition.z = ORBIT_MAX_RADIUS * fYScale * (FLOAT)cos(t);
D3DVECTOR vVelocity;      //速度向量，x，y，z分量
vVelocity.x = ORBIT_MAX_RADIUS * fXScale * (FLOAT)sin(t+0.05f);
vVelocity.y = 0.0f;
vVelocity.z = ORBIT_MAX_RADIUS * fYScale * (FLOAT)cos(t+0.05f);
memcpy( &g_dsBufferParams.vPosition, &vPosition, sizeof(D3DVECTOR) );
memcpy( &g_dsBufferParams.vVelocity, &vVelocity, sizeof(D3DVECTOR) );
```

```
        UpdateGrid( vPosition.x, vPosition.z );
    if( g_pDS3DBuffer )
        g_pDS3DBuffer->SetAllParameters( &g_dsBufferParams, DS3D_IMMEDIATE );
    }                              //设置所有参数
```

此段代码实现定时器功能，对声源位置向量和速度向量进行设置。同时触发 DirectSound 声源的运动。

STEP 12 创建波形文件。

定义波形文件 CWaveFile 类，此封装了许多相关函数，封装图如图 5-8 所示。

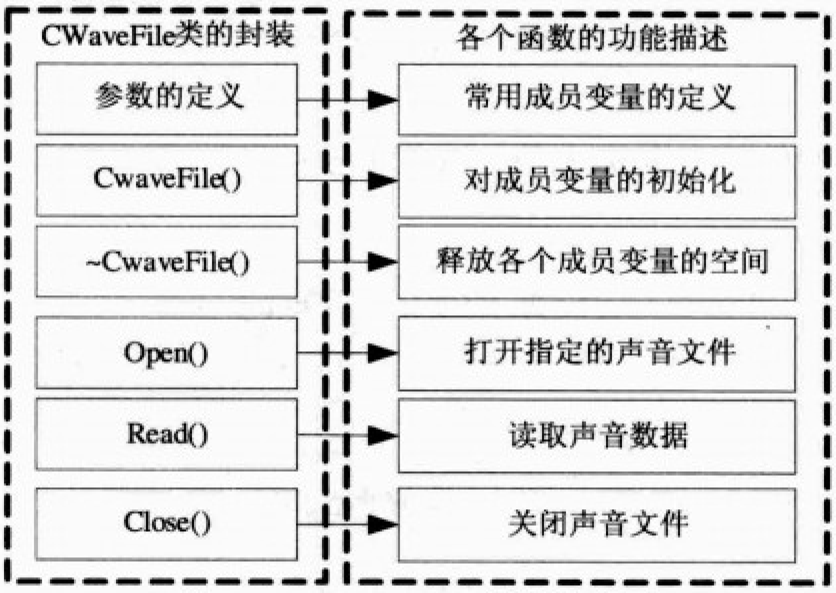


图 5-8 CWaveFile 类的封装图

下面是具体函数的实现代码。

```
class CWaveFile
{
public:
    WAVEFORMATEX* m_pwfx;          //WAVEFORMATEX 结构的指针
    HMMIO         m_hmmio;         //WAVE的MM I/O句柄
    MMCKINFO      m_ck;            //RIFF数据块
    MMCKINFO      m_ckRiff;        //用于打开WAVE文件
    DWORD         m_dwSize;        //WAVE文件大小
    DWORD         m_dwFlags;       //标志位
    BOOL          m_bIsReadingFromMemory;
    BYTE*         m_pbData;
    BYTE*         m_pbDataCur;     //数据指针
    ULONG         m_ulDataSize;    //数据大小
    TCHAR*        m_pResourceBuffer;
protected:
   HRESULT ReadMMIO();
   HRESULT WriteMMIO( WAVEFORMATEX *pwfxDest ); //从内存映射I/O读取数据
public:
    CWaveFile();                   //构造函数
    ~CWaveFile();                  //析构函数
    HRESULT Open( LPTSTR strFileName, WAVEFORMATEX* pwfx,
DWORD dwFlags );                   //打开函数
```

```
    HRESULT Close();                        //关闭函数
    HRESULT Read( BYTE* pBuffer, DWORD dwSizeToRead,
DWORD* pdwSizeRead );                       //写入数据函数
    HRESULT Write( UINT nSizeToWrite, BYTE* pbData, UINT* pnSizeWrote );//读数据
    DWORD  GetSize();                       //获取数据大小函数
    HRESULT ResetFile();                    //重置文件
    WAVEFORMATEX* GetFormat() { return m_pwfx; }; //获取文件格式
};
```

此段代码创建了一个波形文件类，此类封装了播放声音时的操作函数及一些成员变量。如打开、关闭函数，读取、写入数据函数等。

STEP 13 构造函数。

```
CWaveFile::CWaveFile()                      //初始化成员变量
{
    m_pwfx = NULL;                          //指向波形文件
    m_hmmio = NULL;                         //音频文件句柄
   m_pResourceBuffer = NULL;                //指向资源缓冲区
    m_dwSize = 0;                           //代表缓冲区的大小
    m_bIsReadingFromMemory = FALSE;         //从内存读取数据的变量
}
```

构造函数主要实现了成员变量的初始化。

STEP 14 析构函数。

```
CWaveFile::~CWaveFile()
{
    Close();                                //关闭波形文件
    if( !m_bIsReadingFromMemory )
        SAFE_DELETE_ARRAY( m_pwfx );        //删除缓冲区指针数组
}
```

析构函数主要用来释放各个成员变量的空间，并销毁对象。

STEP 15 打开声音文件。

```
HRESULT CWaveFile::Open( LPTSTR strFileName, WAVEFORMATEX* pwfx,
DWORD dwFlags )
{
    HRESULT hr;
    m_dwFlags = dwFlags;                    //标志位
    m_bIsReadingFromMemory = FALSE;
    if( m_dwFlags == WAVEFILE_READ )        //以只读方式打开文件
    {
        if( strFileName == NULL )
            return E_INVALIDARG;
        SAFE_DELETE_ARRAY( m_pwfx );        //若打开失败，则删除缓冲区指针数组
   m_hmmio = mmioOpen( strFileName, NULL, MMIO_ALLOCBUF | MMIO_READ );
        if( NULL == m_hmmio )
        {
```

```
            //若加载失败，则重试
            if(NULL==(hResInfo=FindResource(NULL,strFileName,
TEXT("WAVE") ) ) )
            {
              if(NULL==(hResInfo=FindResource(NULL,strFileName,
TEXT("WAV") ) ) )
                    return E_FAIL;
            }
            MMIOINFO mmioInfo;
            ZeroMemory( &mmioInfo, sizeof(mmioInfo) );
            mmioInfo.fccIOProc = FOURCC_MEM;
            mmioInfo.cchBuffer = dwSize;          //文件缓冲区的字节大小
            mmioInfo.pchBuffer = (TCHAR*) m_pResourceBuffer;
            m_hmmio=mmioOpen(NULL,&mmioInfo,     //以只读方式开发文件
 MMIO_ALLOCBUF | MMIO_READ );
        }
        if( FAILED( hr = ReadMMIO() ) )
        {
            mmioClose( m_hmmio, 0 );              //如果不是WAVE文件，则读取将失败
            return hr ;
        }
        if( FAILED( hr = ResetFile() ) )
            return  hr ;
            m_dwSize = m_ck.cksize;               //重设后，WAV文件重新恢复
    }
```

此段代码实现打开声音文件功能。如果是声音文件，则打开声音文件，否则打开失败。

STEP 16 读取文件内容。

```
HRESULT CWaveFile::ReadMMIO()
{
    MMCKINFO       ckIn;                           //块信息
    PCMWAVEFORMAT  pcmWaveFormat;                  //定义结构对象
    m_pwfx = NULL;
    if( ( 0 != mmioDescend( m_hmmio, &m_ckRiff, NULL, 0 ) ) )
        return  E_FAIL ;
    //检查打开文件是否是声音文件
    if( (m_ckRiff.ckid != FOURCC_RIFF) ||
        (m_ckRiff.fccType != mmioFOURCC('W', 'A', 'V', 'E') ) )
        return E_FAIL ;
    //寻找‘fmt’块
    ckIn.ckid = mmioFOURCC('f', 'm', 't', ' ');
    if( 0 != mmioDescend( m_hmmio, &ckIn, &m_ckRiff, MMIO_FINDCHUNK ) )
        return  E_FAIL ;
     //读取‘fmt’块并加载到定义的结构体
    if( mmioRead( m_hmmio, (HPSTR) &pcmWaveFormat,
                  sizeof(pcmWaveFormat)) != sizeof(pcmWaveFormat) )
```

```
        return E_FAIL ;
    if( pcmWaveFormat.wf.wFormatTag == WAVE_FORMAT_PCM )
    {
        m_pwfx = (WAVEFORMATEX*)new TCHAR[ sizeof(WAVEFORMATEX) ];
        if( NULL == m_pwfx )
            return  E_FAIL ;
        //从PCM结构体读取数据加载到声音文件中
        memcpy( m_pwfx, &pcmWaveFormat, sizeof(pcmWaveFormat) );
        m_pwfx->cbSize = 0;
    }
    else
    {
        if( NULL == m_pwfx )
            return  E_FAIL ;
        //从pcm结构体中复制数据到waveformatex结构体中
        memcpy( m_pwfx, &pcmWaveFormat, sizeof(pcmWaveFormat) );
        m_pwfx->cbSize = cbExtraBytes;
        //现在读取这些字节到结构体中
    if( mmioRead( m_hmmio, (TCHAR*)(((BYTE*)&(m_pwfx->cbSize))+sizeof(WORD)),
                cbExtraBytes ) != cbExtraBytes )
        {
            SAFE_DELETE( m_pwfx ); //删除对象
            return  E_FAIL ;
        }
    }
    //把'fmt'块输入到文件中.
    if( 0 != mmioAscend( m_hmmio, &ckIn, 0 ) )
    {
        SAFE_DELETE( m_pwfx );      //删除对象
        return  E_FAIL ;
    }
    return S_OK;
}
```

此段代码实现读取声音数据的功能。首先读取声音文件数据的块信息，然后寻找并读取数据块，最后把数据输出并删除对象。

STEP 17 关闭文件。

```
HRESULT CWaveFile::Close()
{
    if( m_dwFlags == WAVEFILE_READ )
    {
        mmioClose( m_hmmio, 0 );   //关闭音频文件
        m_hmmio = NULL;            //音频文件句柄
        SAFE_DELETE_ARRAY( m_pResourceBuffer );     //关闭缓冲区指针数据
    }
    else
```

```
    {
        m_mmioinfoOut.dwFlags |= MMIO_DIRTY;
        if( m_hmmio == NULL )
            return CO_E_NOTINITIALIZED;
        if( 0 != mmioSetInfo( m_hmmio, &m_mmioinfoOut, 0 ) )
            return E_FAIL ;
            //跳出‘data’数据块
        if( 0 != mmioAscend( m_hmmio, &m_ck, 0 ) )
            return E_FAIL ;
            //跳出RIFF块
        if( 0 != mmioAscend( m_hmmio, &m_ckRiff, 0 ) )
            return E_FAIL ;
        mmioSeek( m_hmmio, 0, SEEK_SET );          //定位文件的当前位置
        if( 0 != (INT)mmioDescend( m_hmmio, &m_ckRiff, NULL, 0 ) )
            return E_FAIL ;                        //跳出RIFF块
        if( 0 == mmioDescend( m_hmmio, &m_ck, &m_ckRiff, MMIO_FINDCHUNK ) )
        {
            DWORD dwSamples = 0;
            mmioWrite( m_hmmio, (HPSTR)&dwSamples, sizeof(DWORD) );
                                                   //向指定的文件中写入数据
            mmioAscend( m_hmmio, &m_ck, 0 );       //离开数据块
        }
    }
    return S_OK;
}
```

此段代码实现了退出声音文件、关闭声音文件功能。首先关闭声音文件，然后关闭缓冲区，跳出 RIFF 块、data 块和 fmt 块。

5.4.3 程序运行结果

具体的程序代码已在上面设计完成，下面就要验证这些程序是否正确，并显示系统演示效果。

1. 编译和运行

选择“生成”菜单下的“生成解决方案”命令或按“F7”键，系统将会逐步完成编译和链接，生成可执行文件。当界面的下方“输出”窗口中出现：

========== 生成: 1 已成功, 0 已失败, 0 最新, 0 已跳过==========

则显示程序编译运行正确。

若编译和运行出现错误，则查看最下方“输出”窗口中所出现的错误提示，在调试错误处双击错误信息所在行，则系统会自动回到错误出现的代码行，然后调试程序，本程序已编译运行正确。

2. 运行程序

STEP 01 选择“调试”→“开始执行（不调试）”选择，运行本程序。本实例的程序运行界面如图 5-9 和图 5-10 所示。

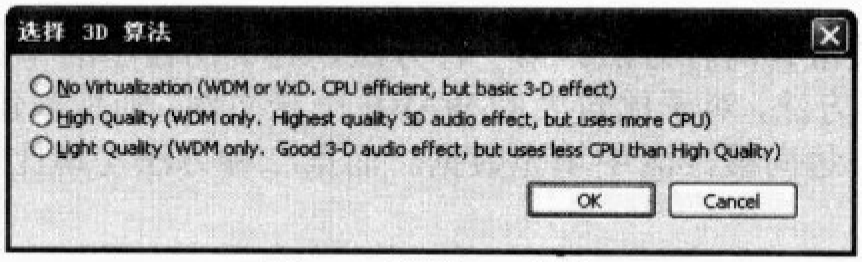


图 5-9 选择 3D 算法

当打开所需的声音文件时，会让用户选择相应的 3D 算法，有 3 种算法可供选择。

- 没有真实的效果（窗口驱动模式或虚拟设备驱动，使用 CPU，但有基本的 3D 效果）。
- 高质量效果（仅有窗口驱动模式，最高质量的 3D 音效，但是占用了较多 CPU）。
- 次高质量效果（仅有窗口驱动模式，好的 3D 音效，比高质量效果占用较少 CPU）。

根据自己的需要选择所需的 3D 算法，如图 5-9 所示。

STEP 02 单击“打开文件”按钮，打开声音文件，再单击“播放”按钮，即开始播放，并且在状态文本框中显示当前的播放状态。可以在声音属性中通过滑动条调整最大距离、最小距离、多普勒因子和滚降因子。并且在声源运行的图片中可以看到声源的运动状态。

当用户选择了不同的 3D 算法，或者设置了声音属性中的最大距离、最小距离、多普勒因子和滚降因子时，可以听到不同的 3D 效果（由远到近、由近到远等），同时也可以看到声源会按照设置的值沿着坐标轴运动，如图 5-10 所示。

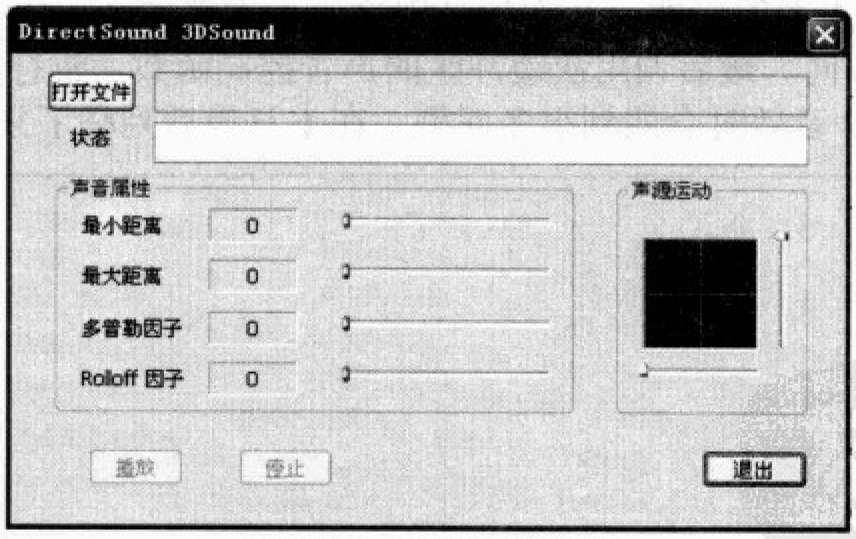


图 5-10 程序运行界面

5.5 DirectSound 3D开发常见问题

在用 DirectSound 实现 3D 音效时，经常会遇到许多问题，下面列举了几个常见的问题和解决方法。

1）3D 音效支持多通道的声音文件吗

这个读者必须清楚，3D 音效只支持单声道的 WAVE 声音文件，一个多声道的 WAVE 文件无法实现三维立体声的效果，就是用 3D 音效被不支持 3D 音效。

2）播放音频文件通常有几种方法

播放音频文件，通常有两种方法。第一种方法，对于比较小的 WAVE 声音文件，用静态缓冲区来播放；第二种方法，对于比较大的 WAVE 声音文件，则是创建一个比较小的缓冲区，一边播放声音数据，一边向缓冲区中填充数据。而第二种方法实现比第一种方法困难，需要另起一个线程。

3）怎样对 3D 缓冲区参数进行批量设置

应用程序可以单独地或是批量地得到和设置 3D 声音缓冲区的参数。如果发声者（声源）移动，而应用程序要立即设置多个参数，比如位置、方向和速度等，则可以调用 IDirectSound3DBuffer8::GetAllParameters 取回当前的参数，然后改变 DS3DBUFFER 结构体里的参数，再把结构传到 IDirectSound3DBuffer::SetAllParameters 函数里，这样就完成了设置过程。

5.6 本章小结

本章用一个语音 3D 特效器的实例详细介绍了如何用 DirectSound 实现特殊 3D 音效。3D 音效的原理，实际就是根据发声者和听者的位置、声音速度和发生锥区域等信息，计算听者的音强，以决定播放的声音大小。使用 3D 音效播放任意一个 WAVE 文件，需要先创建一个次缓冲区，同时将 WAVE 文件的声音数据加入到次缓冲区。通过次缓冲区创建一个发声者对象（声源），并设置发声者的属性。其次就是创建一个主缓冲区，由它创建一个听者对象，并设置听者的位置和方向。最后调用次缓冲区的声音控制函数，进行播放、停止等操作。在用 DirectSound 实现 3D 音效时会遇到很多问题，在本章最后列举了常见的问题和解决方法。

第6章 对拾音设备录制自己的语音文件

在 DirectSound 所能做的事情中，有一个很重要的功能就是可以从麦克风或者其他拾音设备中捕获声音，并且可以保存为 WAVE 文件。本章将通过一个案例来认识和了解 DirectSound 的这一功能，通过拾音设备来录制自己的语音文件。首先对此案例进行分析和设计，然后通过对所创建的缓冲区进行管理，从而得到录制的语音文件。下面进行详细的介绍。

本章重点

- 系统分析和设计
- 管理捕获缓冲区
- 系统效果演示
- DirectSound 录制语音文件开发常见问题

6.1 系统分析和设计

本案例实现了用 DirectSound 的拾音设备来录制自己的语音文件。可以实现从声卡、麦克风或者其他的拾音设备获得的声音数据进行捕获，然后进行了录制，最后保存为 WAVE 格式的文件。

前面已经讲到 DirectSound 的一个很重要的功能就是可以从麦克风或者其他的拾音设备捕获、录制声音并保存为 WAV 文件。其主要的设计思路，就是先根据选择的拾音（录音）设备来创建设备对象，然后通过所选择的设备对象创建一个捕获缓冲区对象；启动缓冲区，开始录音，拾音设备将把所录制的声音数据写入缓冲区，应用程序可以从缓冲区把数据读取出来写文件，最后把录制的声音数据保存为 WAVE 文件即可，这就实现了录音功能。具体的设计流程如图 6-1 所示。

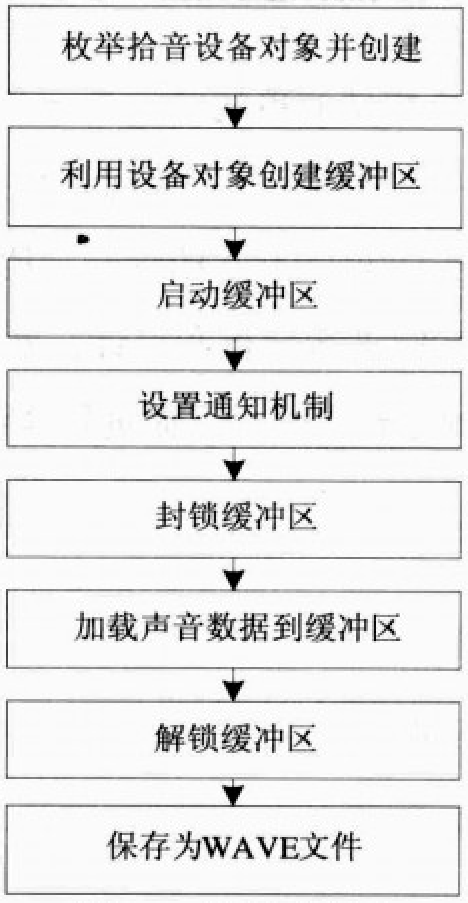


图 6-1　设计流程图

利用 DirectSound 进行录音要用到的 3 个非常重要的对象如下。

- IDirectSoundCapture8：设备对象，根据用户录音的设备创建的设备对象，利用该对象可以获取设备的属性。
- IDirectSoundCaptureBuffer8：缓冲区对象，该对象由设备对象创建，主要用来操作音频数据。
- IDirectSoundNotify8：事件通知对象，该对象用来通知应用程序从缓冲区中将数据取走，写入文件保存起来。

6.2 管理捕获缓冲区

利用 DirectSound 的捕获声音并保存为 WAV 文件的功能，实现录制自己的语音文件，必须要创建捕获缓冲区，来保存从拾音设备（麦克风或其他声音输入设备）捕获的声音数据，为保存为 WAV 文件做准备。下面就来介绍如何创建和管理捕获缓冲区。

6.2.1 创建捕获缓冲区

在创建设备前要先枚举录音的设备，如果应用程序就是想从用户默认的设备上进行声音的录制，那么就没有必要枚举系统中的所有录音设备，当用户调用 DirectSoundCaptureCreate8 或者另外一个函数 DirectSoundFullDuplexCreate8 的时候，其实就指定了一个默认的（拾音设备）录音设备。当然，在下面的情况下，用户就必须要枚举系统中所有的设备。

- 应用程序没有能力利用所有的设备。
- 应用程序要求 2 个或者更多的设备。
- 希望提供给用户设备的选择。

枚举设备有 3 个目的。

- 报告什么硬件可用。
- 它对每个设备都提供一个 GUID。
- 被枚举后，它让用户为每个设备创建一个暂时的设备对象，以便能检查设备性能。

为了枚举设备，必须首先建立一个回调函数，这个回调函数可以被系统中的每个设备调用。在这个函数中用户可以做任何想做的事情，这个函数的命名也没有任何的限制，但是函数应该以 DSEnumCallback 为原型。如果枚举继续，则回调函数必须返回 TRUE，如果枚举结束，例如用户找到合适的设备，则这个函数就要返回 FALSE。枚举设备通常要调用 DirectSoundCaptureEnumerate 函数，如下：

```
DWORD pv;
HRESULT hr = DirectSoundCaptureEnumerate(  //定义枚举设备
    (LPDSENUMCALLBACK)DSEnumProc, (VOID*)&pv);
```

第一个参数是回调函数的地址指针，第二个参数可以是任何 32 位值，它是在回调函数要得到的值。

注意：

被枚举的第一个设备总是被主声音捕获设备所命令，并且回调函数的 lpGUID 参数必须是 NULL。在控制模板中用户所设置的对象代表首选的捕获设备。当用户选择一个设备时，枚举此设备，并且应用程序可以把它加载到主声音捕获设备。主缓冲区捕获设备用合适的名字和 GUID 也被枚举。

1．创建设备对象

用户可以通过 DirectSoundCaptureCreate8 或者 DirectSoundFullDuplexCreate8 函数直接创建设备对象，该函数返回一个指向 IDirectSoundCapture8 接口的指针。

DirectSoundCaptureCreate 的 lpcGUID 参数，或者是 DirectSoundFullDuplexCreate8 的 pcGuidRenderDevice 参数，可能是通过枚举所获得的一个 GUID，或者是表 6-1 中所预先定义的 GUIDs 之一。

表 6-1　GUID 描述

GUID	描述
DSDEVID_DefaultCapture	系统默认的捕获设置。通过一个空指针来指定此设备来代替一个设备的 GUID
DSDEVID_DefaultVoiceCapture	默认的声音交换捕获设备。它是一个次要的设备，例如带 USB 接口的麦克风等

如果没有设备驱动被声明，则返回失败。也可以用 CoCreateInstance 函数来创建一个对象，步骤和设备对象相似。如果用 CoCreateInstance，则这个对象被默认的捕获设备所创建。下面是 CoCreateInstance 函数的原型。

```
STDAPI CoCreateInstance(
    REFCLSID rclsid,         //创建的COM对象的类标识符(CLSID)
    LPUNKNOWN pUnkOuter,     //指向接口IUnknown的指针
    DWORD dwClsContext,      //运行可执行代码的上下文
    REFIID riid,             //创建的COM对象的接口标识符
    LPVOID * ppv             //用来接收指向COM对象接口地址的指针变量
);
```

函数的主要功能是用指定的类标识符创建一个 COM 对象，用指定的类标识符创建一个未初始化的对象。

如果想使录音重放和捕获设备对象共存，则应该在创建和初始化捕获设备对象之前创建并初始化重复录音重放设备对象，或者用 DirectSoundFullDuplexCreate8。在默认的全双工通信音频下不能配置一些音频设备。如果应用程序在创建和初始化重放设备对象和捕获设备对象时出现问题，那么就应该检查音频设备属性以确保全双工通信。

2．捕获设备性能

可以通过 IDirectSoundCapture8::GetCaps 方法来获取录音的性能，这个函数的参数是一个 DSCCAPS 类型的结构，在传递这个参数之前，一定要初始化该结构的 dwSize 成员变量。同时，用户可以通过这个结构返回设备支持的声道数，以及类似于 WAVEINCAPS 结构的其他设备属性。

在非 Windows 驱动模式（WDM）下，一个设备通常每次仅能被一个应用程序所用。如果驱动器同时支持更多的设备，则更多的应用程序将会用到它，在 DSCCAPS 结构中的成员变量 dwFlags 中 DSCCAPS_MULTIPLECAPTURE 被返回。每个应用程序为捕获缓冲区能设置它自己的格式。

下面介绍一下上面讲到的 IDirectSoundCapture8::GetCaps 函数，此函数原型为：

```
HRESULT GetCaps(
  LPDSCCAPS pDSCCaps  //指向结构体的指针
);
```

此函数的功能是获取捕获设备的性能。参数 pDSCCaps 是结构体的指针，来获取捕获设备的信息。

3. 创建捕获缓冲区

可以通过 IDirectSoundCapture8::CreateCaptureBuffer 来创建一个录音的缓冲区对象，这个函数的一个参数采用 DSCBUFFERDESC 类型的结构来说明缓冲区的一些特性，这个结构的最后一个成员变量是一个 WAVEFORMATEX 结构，这个结构一定要初始化成你需要的 WAV 格式。

说明一下，如果用户的应用程序一边播放声音一边录制声音，或者录制的缓冲区格式和主缓冲区不一样，那么创建录制缓冲区对象就会失败。原因就是有些声卡只支持一种时钟，不能同时支持录音和播放。下面的函数演示了如何创建一个录音的缓冲区对象。

```
HRESULT CreateCaptureBuffer(LPDIRECTSOUNDCAPTURE8 pDSC,
                            LPDIRECTSOUNDCAPTUREBUFFER8* ppDSCB8)
{
 HRESULT hr;
 DSCBUFFERDESC  dscbd;                          //创建结构体对象
 LPDIRECTSOUNDCAPTUREBUFFER  pDSCB;             //指向捕获缓冲区的指针
 WAVEFORMATEX  wfx ={WAVE_FORMAT_PCM, 2, 44100, 176400, 4, 16, 0};
 //分别初始化格式类型、声道数、采样率、数据率、采样块对齐大小、位数、块大小
  if ((NULL == pDSC) || (NULL == ppDSCB8)) return E_INVALIDARG;
 dscbd.dwSize = sizeof(DSCBUFFERDESC);
 dscbd.dwFlags = 0;                             //缓冲区的标志位
 dscbd.dwBufferBytes = wfx.nAvgBytesPerSec;     //缓冲区的大小
 dscbd.dwReserved = 0;                          //保留未用，必须为0
 dscbd.lpwfxFormat = &wfx;                      //缓冲区格式属性
 dscbd.dwFXCount = 0;
 dscbd.lpDSCFXDesc = NULL;
 if (SUCCEEDED(hr = pDSC->CreateCaptureBuffer(&dscbd, &pDSCB, NULL)))
                                                //创建捕获缓冲区
 {                                              //获取DirectSoundCaptureBuffer接口
  hr=pDSCB->QueryInterface(IID_IDirectSoundCaptureBuffer8, (LPVOID*)ppDSCB8);
  pDSCB->Release();                             //释放捕获
 }
 return hr;
}
```

此函数实现了创建一个录音缓冲区对象。这个缓冲区对象能够处理 1 秒钟的数据。

注意：

> 所传递的录音设备对象参数必须通过 DirectSoundCaptureCreate8 来创建，而不是 DirectSoundCaptureCreate 接口，否则缓冲区对象不支持 IDirectSoundCaptureBuffer8 接口。

4．获取录音缓冲区信息

可以通过 IDirectSoundCaptureBuffer8::GetCaps 方法来获取录音缓冲区的大小，但一定要在传递参数以前初始化 DSCBCAPS 结构类型参数的 dwSize 成员变量。

为了获取缓冲区中数据的格式，可以通过 IDirectSoundCaptureBuffer8::GetFormat 方法来获取缓冲区中的数据格式，此方法通过 WAVEFORMATEX 结构返回音频数据的信息。为了要知道一个捕获缓冲区目前是否工作，可以通过 IDirectSoundCaptureBuffer8::GetStatus 来获取，这个函数通过一个 DWORD 类型的参数来表示该缓冲区是否正在录音。

IDirectSoundCaptureBuffer8::GetCurrentPosition 方法可以获取缓冲区中读指针和录制指针的偏差。读指针指向填充到该缓冲区中的数据的末端，录制指针则指向复制到硬件的数据的末端，读指针指向的前段数据都是安全数据，都可以安全地复制。

下面介绍上面所讲到的 IDirectSoundCaptureBuffer8::GetFormat、IDirectSound Capture Buffer8::GetStatus、IDirectSoundCaptureBuffer8::GetCurrentPosition 函数。

IDirectSoundCaptureBuffer8::GetFormat 函数的原型为：

```
HRESULT GetFormat(
  LPWAVEFORMATEX pwfxFormat,        //指向WAVEFORMATEX结构体的指针
  DWORD dwSizeAllocated,            //WAVEFORMATEX结构体大小
  LPDWORD pdwSizeWritten            //指向写入WAVEFORMATEX结构体数据的指针
);
```

此函数用来捕获缓冲区的声音文件。

IDirectSoundCaptureBuffer8::GetStatus 函数的原型为：

```
HRESULT GetStatus(
  LPDWORD pdwStatus                 //获取捕获缓冲区的变量地址
);
```

此函数用来获取捕获缓冲区状态。

IDirectSoundCaptureBuffer8::GetCurrentPosition 函数的原型为：

```
HRESULT GetCurrentPosition(
  LPDWORD pdwCapturePosition,       //指向捕获缓冲区位置的指针
  LPDWORD pdwReadPosition           //指向读取位置的指针
);
```

此函数用来获取读取缓冲区的位置。

5．录音缓冲区的通知机制

为了安全、定期地从录音缓冲区中复制数据，应用程序必须要知道何时读指针指向了特定的位置，一个得到此信息的方法就是通过 IDirectSoundCaptureBuffer8::GetCurrentPosition 方法来获取读指针的位置，另外一个更有效的方法就是采用通知机制，通过 IDirectSoundNotify8::SetNotificationPositions 方法，可以设置任何一个小于缓冲区的位置来触发一个事件。但是，当缓冲区正在运行的时候，不要设置。下面是 IDirectSoundNotify8:: SetNotificationPositions 函数的原型。

```
HRESULT SetNotificationPositions(
  DWORD dwPositionNotifies,        //DSBPOSITONNOTIFY结构体的数量
  LPCDSBPOSITIONNOTIFY pcPositionNotifies
                                   //指向DSBPOSITIONNOTIFY结构体数据
);
```

此函数的主要功能是设置通知位置。

如何来设置一个触发事件呢，首先要获得 IDirectSoundNotify8 接口指针，可以通过缓冲区对象的 QuerInterface 方法来获取这个指针接口，对于指定的任何一个通知位置，都要通过 CreateEvent 方法来创建一个 Win 32 对象，然后将此对象的句柄传给 DSBPOSITIONNOTIFY 结构的 hEventNotify 成员，通过该结构的 dwOffset 成员来指定需要通知的位置在缓冲区中的偏移量。

最后将这个结构的地址或者结构数组的地址传递给 SetNotificationPositions 函数，下面的示例设置了 3 个通知。

```
HRESULT SetCaptureNotifications(LPDIRECTSOUNDCAPTUREBUFFER8 pDSCB)
{
LPDIRECTSOUNDNOTIFY8 pDSNotify;        //通知事件指针
WAVEFORMATEX wfx;                      //结构体对象
HANDLE rghEvent[cEvents] = {0};
DSBPOSITIONNOTIFY rgdsbpn[cEvents];    //创建结构体对象
HRESULT hr;
if (NULL == pDSCB) return E_INVALIDARG;
if (FAILED(hr = pDSCB->QueryInterface(IID_IDirectSoundNotify,
(LPVOID*)&pDSNotify)))                 //获取DirectSoundNotify接口
{
return hr;
}                                      //获取声音文件格式
if (FAILED(hr = pDSCB->GetFormat(&wfx, sizeof(WAVEFORMATEX), NULL)))
{
return hr;
}
for (int i = 0; i < cEvents; ++i)      //创建事件
{
rghEvent[i] = CreateEvent(NULL, TRUE, FALSE, NULL);
     if (NULL == rghEvent[i])
{
hr = GetLastError();
return hr;
}
}
rgdsbpn[0].dwOffset = (wfx.nAvgBytesPerSec/2) -1;      //描述通知
rgdsbpn[0].hEventNotify = rghEvent[0];                 //第一个事件通知
rgdsbpn[1].dwOffset = wfx.nAvgBytesPerSec - 1;
rgdsbpn[1].hEventNotify = rghEvent[1];                 //第二个事件通知
rgdsbpn[2].dwOffset = DSBPN_OFFSETSTOP; .              //通知触发时缓冲区的偏移
```

```
rgdsbpn[2].hEventNotify = rghEvent[2];                    //第三个事件通知
hr = pDSNotify->SetNotificationPositions(cEvents, rgdsbpn);  //设置通知位置
pDSNotify->Release();                                       //释放对象
return hr;
}
```

上面的示例实现了 3 个事件通知。当位置达到缓冲区的一半时就会触发一个通知事件，当达到缓冲区的末端时，触发第二个通知事件，当录音停止时触发第三个通知事件。

6. 将录音写入 wav 文件

WAV 文件是采用 RIFF 格式的文件，其中包含一系列的块结构用来描述头信息和数据信息。Win 32 API 提供了打开和关闭 RIFF 文件的函数功能、寻找块等。这些函数都是以“mmio”开始的。

Directsound API 并不包括读写 WAV 格式文件的函数。但是，在 Directsound 中封装了一个 CWaveFile 类用来管理录制的 WAV 文件。CWaveFile 类通过下面 3 种方法来管理所录制的声音文件。

- 通过打开函数来写入文件的头信息。
- 通过写函数来写入文件的数据。
- 关闭函数写入文件的长度和关闭文件。

保存 WAV 文件的第一步就是调用 CWaveFile::Open 方法。用此方法来创建文件和写入 WAV 格式的数据。参数是文件名，初始化 WAVEFORMATEX 结构的指针和 WAVEFILE_WRITE 标志位，此方法返回 HRESULT。下面是代码，打开文件并创建一个 WAV 格式的文件。

```
CWaveFile g_pWaveFile;
WAVEFORMATEX wfxInput;                          //创建对象
ZeroMemory( &wfxInput, sizeof(wfxInput));
wfxInput.wFormatTag = WAVE_FORMAT_PCM;         //文件格式类型
wfxInput.nSamplesPerSec = 22050                //采样率
wfxInput.wBitsPerSample = 8;                   //位数
wfxInput.nChannels = 1;                        //通道数
wfxInput.nBlockAlign =4                        //块对齐大小
wfxInput.nChannels * (wfxInput.wBitsPerSample / 8);
wfxInput.nAvgBytesPerSec =                     //数据率
wfxInput.nBlockAlign * wfxInput.nSamplesPerSec;
g_pWaveFile = new CWaveFile;                   //申请空间
if (FAILED(g_pWaveFile->Open("mywave.wav", &wfxInput,
WAVEFILE_WRITE)))                              //打开文件
{
g_pWaveFile->Close();
}
```

此段代码演示的是创建一个 WAV 格式的文件并设置文件的格式，最终打开此文件。

下面的步骤就是如何从捕获缓冲区对象中将 WAV 数据保存到文件中。

6.2.2 启动缓冲区

通过调用 IDirectSoundCaptureBuffer8::Start 方法使缓冲区对象开始工作，通过给这个函数 dwFlags 传递一个 DSCBSTART_LOOPING 参数，缓冲区就会不停地工作，而不是当缓冲区填充满了就停止工作，当缓冲区满了之后，就会从头重新填充。

等待所期望的事件通知，当缓冲区被填充到某个用户期望的位置时，就会触发通知。

```
HRESULT RecordCapturedData()
{
HRESULT hr;
VOID* pbCaptureData = NULL;  //捕获数据
DWORD dwCaptureLength;       //捕获长度
VOID* pbCaptureData2 = NULL; //捕获数据
DWORD dwCaptureLength2;      //捕获长度
VOID* pbPlayData = NULL;     //播放数据
UINT dwDataWrote;
DWORD dwReadPos;             //读写位置
LONG lLockSize;              //锁存大小
if (NULL == g_pDSBCapture)
return S_FALSE;
if (NULL == g_pWaveFile)
return S_FALSE;
if (FAILED (hr = g_pDSBCapture->GetCurrentPosition(   //获取当前读位置
NULL, &dwReadPos)))
return hr;
```

此段代码演示了通过参数设置来获取读写位置。

6.2.3 封锁缓冲区

当收到通知时，就要调用 IDirectSoundCaptureBuffer8::Lock 来锁住缓冲区的一部分，但是不要将捕获指针所指向的内存锁住，可以调用 IDirectSoundCaptureBuffer8::GetCurrentPosition 方法来获取读指针的位置。在传递给 Lock 函数的参数中，一定要指定内存的大小和偏移量，这个函数会返回用户锁住的内存的起始地址及块数据的大小。

函数 DirectSoundCaptureBuffer8::Lock 函数的原型：

```
HRESULT Lock(
  DWORD dwOffset,               //偏移量
  DWORD dwBytes,                //锁存缓冲区的大小
  LPVOID *ppvAudioPtr1,         //缓冲区中第一个被锁存部分的变量指针
  LPDWORD pdwAudioBytes1,       //块字节数的变量地址
  LPVOID *ppvAudioPtr2,         //缓冲区中第二个被锁存部分的变量指针
  LPDWORD pdwAudioBytes2,       //块字节数的变量地址
```

```
  DWORD dwFlags                    //缓冲区标志位
);
```

此函数实现锁存缓冲区，下面来锁存捕获缓冲区。

```
//锁存捕获缓冲区
lLockSize = dwReadPos - g_dwNextCaptureOffset;
if( lLockSize < 0 ) lLockSize += g_dwCaptureBufferSize;
if( lLockSize == 0 ) return S_FALSE;
if (FAILED(hr = g_pDSBCapture->Lock(g_dwNextCaptureOffset, lLockSize,
                                                              //锁存缓冲区
&pbCaptureData, &dwCaptureLength,
&pbCaptureData2, &dwCaptureLength2, 0L)))
return hr;
//向缓冲区加载数据
if (FAILED( hr = g_pWaveFile->Write( dwCaptureLength,     //加载数据
(BYTE*)pbCaptureData, &dwDataWrote)))
return hr;
if (pbCaptureData2 != NULL)
{
if (FAILED(hr = g_pWaveFile->Write(                       //加载数据
dwCaptureLength2, (BYTE*)pbCaptureData2,
&dwDataWrote)))
return hr;
}
```

此段代码实现了对缓冲区的锁存，同时实现了向缓冲区加载数据的功能。

6.2.4 解锁缓冲区

从锁住的内存中复制数据。复制完成后要用 IDirectSoundCaptureBuffer8::Unlock 方法来解锁缓冲区。

Lock 方法锁住缓冲区的一部分。把锁住缓冲区的指针返回到此缓冲区，允许应用程序读或写音频数据到内存。DirectSoundCaptureBuffer8::Unlock 方法的原型为：

```
HRESULT Unlock(
  LPVOID pvAudioPtr1,      //缓冲区中第一个被锁存部分的变量指针
  DWORD dwAudioBytes1,     //锁存字节数
  LPVOID pvAudioPtr2,      //缓冲区中第一个被锁存部分的变量指针
   DWORD dwAudioBytes2     //锁存字节数
);
```

此函数实现对缓冲区的锁存功能，下面是调用此方法来解锁缓冲区。

```
g_pDSBCapture->Unlock( pbCaptureData, dwCaptureLength,    //解锁缓冲区
pbCaptureData2, dwCaptureLength2 );
```

此段代码实现对捕获缓冲区的锁存操作。

6.2.5 中止捕获

在停止录音之前，可以反复地重复 6.2.2~6.2.5 节的步骤，如果想停止录音，用户可以调用 IDirectSoundCaptureBuffer8::Stop 方法。

Stop()方法停止缓冲区，从而不再捕获声音数据。如果缓冲区不捕获数据，则此方法将失去作用。Stop 方法的原型：

```
HRESULT Stop();
```

此方法没有参数。如果此方法成功，则返回值为 DS_OK。如果此方法失败，则返回值或许是 DSERR_NODRIVER 和 DSERR_OUTOFMEMORY 两者之一。

此方法包含在 dsound.h 中。

当录音完成时，应用程序关闭 WAV 文件。调用 Close()方法，此方法原型为：

```
g_pWaveFile->Close();    //关闭声音文件
```

Close 函数的原型为：

```
HRESULT Close(void);
```

此方法没有参数，返回一个 HRESULT 类型的值。

6.3 系统效果演示

上节介绍了用 DirectSound 实现捕获声音并保存为 WAV 文件的基本实现原理，下面就来设计具体的实现代码。

6.3.1 声音类的封装

在程序的设计实现之前，首先定义几个类，实现对录制声音函数的定义和管理。图 6-2 是对声音管理类的封装图。

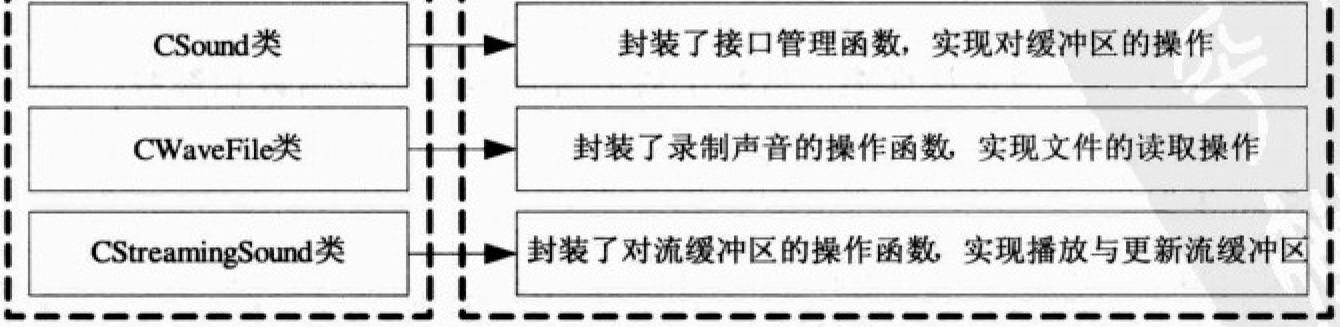


图 6-2 声音管理类的封装

下面是具体的实现代码。

在头文件中添加如下文件之后，就可以方便函数的调用。如果未加入，则在编译时会发生错误。

```
#include <mmsystem.h>
#include <mmreg.h>
#include <dsound.h>
```

宏定义：

```
#define WAVEFILE_READ   1
#define WAVEFILE_WRITE  2
#define DSUtil_StopSound(s)              { if(s) s->Stop(); }
#define DSUtil_PlaySound(s)              { if(s) s->Play( 0, 0 ); }
#define DSUtil_PlaySoundLooping(s)       { if(s) s->Play( 0, DSBPLAY_LOOPING ); }
```

定义 CSoundMananger 类：

```
class CSoundManager
{
protected:
    LPDIRECTSOUND8 m_pDS;
public:
    CSoundManager();                          //构造函数
    ~CSoundManager();                         //析构函数
    HRESULT Initialize( HWND hWnd, DWORD dwCoopLevel );     //初始化函数
    inline  LPDIRECTSOUND8 GetDirectSound() { return m_pDS; }
HRESULT SetPrimaryBufferFormat(DWORD dwPrimaryChannels,      //设置主缓冲区格式
DWORD dwPrimaryFreq,                      //主缓冲区频率
DWORD dwPrimaryBitRate );                 //主缓冲区位率
HRESULTGet3DListenerInterface(LPDIRECTSOUND3DLISTENER* ppDSListener );
HRESULT Create(CSound**ppSound,LPTSTR strWaveFileName,
DWORD dwCreationFlags = 0,
GUID guid3DAlgorithm = GUID_NULL,     //3D算法
DWORD dwNumBuffers = 1 );
HRESULT CreateFromMemory( CSound** ppSound, BYTE* pbData, ULONG ulDataSize,
LPWAVEFORMATEX pwfx, DWORD dwCreationFlags = 0, GUID guid3DAlgorithm = GUID_NULL,
DWORD dwNumBuffers = 1 );
HRESULT CreateStreaming( CStreamingSound**ppStreamingSound, //创建数据流
LPTSTR  strWaveFileName,DWORD  dwCreationFlags,  GUID  guid3DAlgorithm,  DWORD
dwNotifyCount, DWORD dwNotifySize, HANDLE hNotifyEvent );
};
```

此段代码定义了一个声音管理类，封装了对声音管理的一些函数和参数，实现了对缓冲区格式属性的设置。

定义 CSound 类：

```
class CSound
{
protected:
   LPDIRECTSOUNDBUFFER* m_apDSBuffer;                    //指向缓冲区的指针
```

```
    DWORD                   m_dwDSBufferSize;          //缓冲区的大小
    CWaveFile*              m_pWaveFile;               //波形文件
    DWORD                   m_dwNumBuffers;            //缓冲区的数量
    DWORD                   m_dwCreationFlags;         //创建标志位
    HRESULT RestoreBuffer( LPDIRECTSOUNDBUFFER pDSB,
                            BOOL* pbWasRestored );     //恢复缓冲区
public:
    CSound( LPDIRECTSOUNDBUFFER* apDSBuffer,
DWORD dwDSBufferSize, DWORD dwNumBuffers, CWaveFile* pWaveFile,
DWORD dwCreationFlags );
    virtual ~CSound();                                 //析构函数
    HRESULT Get3DBufferInterface(DWORD dwIndex,
         LPDIRECTSOUND3DBUFFER* ppDS3DBuffer );        //获取3D缓冲区接口
    HRESULT FillBufferWithSound(LPDIRECTSOUNDBUFFER pDSB,
                            BOOL bRepeatWavIfBufferLarger );//填充缓冲区
    LPDIRECTSOUNDBUFFER GetFreeBuffer();
    LPDIRECTSOUNDBUFFER GetBuffer( DWORD dwIndex );
    HRESULT Play( DWORD dwPriority = 0, DWORD dwFlags = 0, LONG lVolume = 0,
                LONG lFrequency = -1, LONG lPan = 0 );          //播放函数
    HRESULT Play3D( LPDS3DBUFFER p3DBuffer, DWORD dwPriority = 0,
                                                       //播放3D缓冲区
    DWORD dwFlags = 0, LONG lFrequency = 0 );
     HRESULT Stop();                                   //停止播放函数
    HRESULT Reset();                                   //重置
    BOOL   IsSoundPlaying();                           //声音是否播放
};
```

此类实现对 DirectSound 缓冲区接口的封装和管理功能。实现了缓冲区的创建、删除、声音数据的填充、播放和停止等功能。

定义 CWaveFile 类，从一个 WAVE 文件读或写声音数据。

```
class CWaveFile
{
public:
    WAVEFORMATEX* m_pwfx;          //指向WAVEFORMATEX 结构体的指针
    HMMIO         m_hmmio;         //MM I/O 句柄
    MMCKINFO       m_ck;
    MMCKINFO       m_ckRiff;       //打开一个WAVE文件
    DWORD         m_dwSize;        //声音文件的大小
    MMIOINFO       m_mmioinfoOut;
    DWORD         m_dwFlags;       //标志位
    BOOL         m_bIsReadingFromMemory;
    BYTE*         m_pbData;
    BYTE*         m_pbDataCur;     //数据指针
    ULONG         m_ulDataSize;    //数据大小
    CHAR*         m_pResourceBuffer;
protected:
```

```
    HRESULT ReadMMIO();
    HRESULT WriteMMIO( WAVEFORMATEX *pwfxDest ); //读取结构体信息
public:
    CWaveFile();                    //构造函数
    ~CWaveFile();                   //析构函数
    HRESULT Open(LPTSTR strFileName, WAVEFORMATEX* pwfx,
DWORD dwFlags );
    HRESULT OpenFromMemory(BYTE* pbData,          //打开函数
ULONG ulDataSize, WAVEFORMATEX* pwfx, DWORD dwFlags );
    HRESULT Close();                //关闭函数
    HRESULT Read(BYTE* pBuffer, DWORD dwSizeToRead, DWORD* pdwSizeRead );
    HRESULT Write( UINT nSizeToWrite, BYTE* pbData, UINT* pnSizeWrote );
                                    //写数据
    DWORD  GetSize();               //获取大小
    HRESULT ResetFile();            //重置文件
    WAVEFORMATEX* GetFormat() { return m_pwfx; }; //获取文件格式
};
```

此类封装了许多对声音操作的函数，功能就是对一个声音文件进行读取和写入操作。

定义 CStreamingSound 类：

```
class CStreamingSound : public CSound
{
protected:
    DWORD m_dwLastPlayPos;
    DWORD m_dwPlayProgress;
    DWORD m_dwNotifySize;
    DWORD m_dwNextWriteOffset;
    BOOL  m_bFillNextNotificationWithSilence;
public:
    CStreamingSound( LPDIRECTSOUNDBUFFER pDSBuffer,
DWORD dwDSBufferSize, CWaveFile* pWaveFile,
DWORD dwNotifySize );          // 构造函数用来初始化成员变量
    ~CStreamingSound();        // 析构函数用来释放成员变量的空间
    HRESULT HandleWaveStreamNotification( BOOL bLoopedPlay );
    HRESULT Reset();
};
```

此段代码实现了 DirectSound 在流缓冲区中播放 WAVE 文件的功能。通过 Create()方法加载 WAVE 文件的数据到流缓冲区中并实现播放。

6.3.2 声音管理函数的实现

上面定义了一些类，这些类封装了对声音进行录制的操作函数。下面就对这些操作函数进行说明和设计，图 6-3 是这些函数的实现流程图。

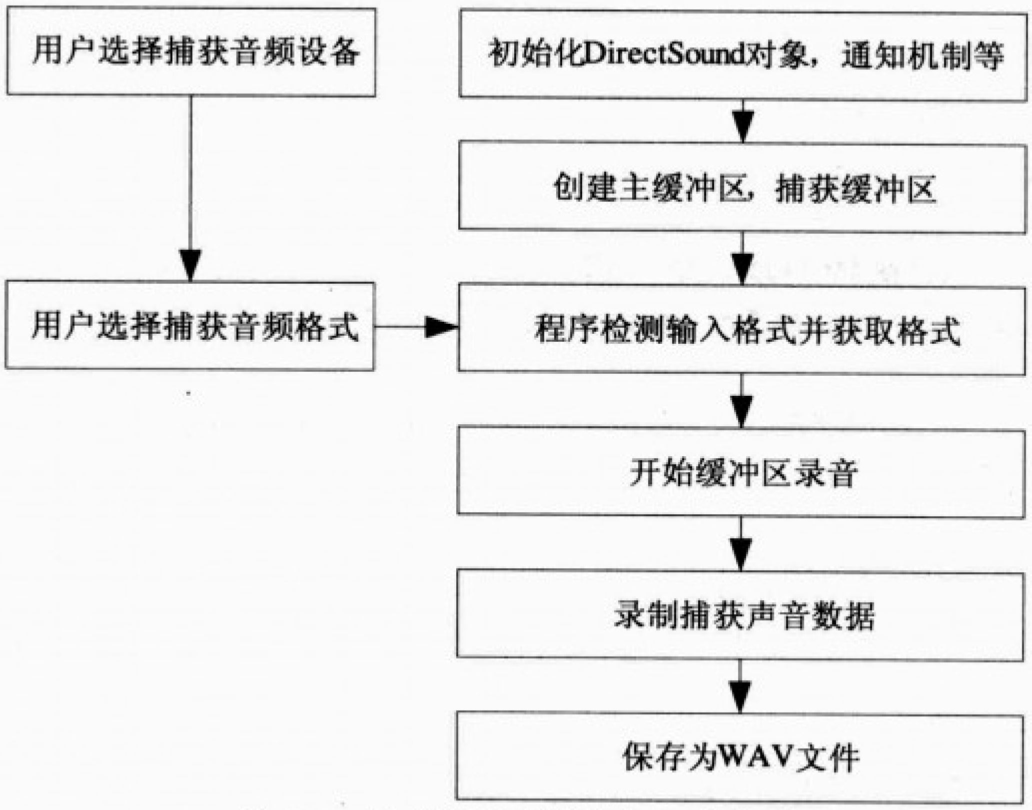


图 6-3　声音管理函数的实现流程图

初始化 IDirectSound 对象：

```
HRESULT CSoundManager::Initialize( HWND  hWnd,
                                   DWORD dwCoopLevel )
{
    HRESULT  hr;
    SAFE_RELEASE( m_pDS );
                                        //创建主声音缓冲区
    if( FAILED( hr = DirectSoundCreate8( NULL, &m_pDS, NULL ) ) )
        return DXTRACE_ERR( TEXT("DirectSoundCreate8"), hr );
      if( FAILED( hr = m_pDS->SetCooperativeLevel( hWnd, dwCoopLevel ) ) )
                                        //设置DirectSound协调级别
        return DXTRACE_ERR( TEXT("SetCooperativeLevel"), hr );
    return S_OK;
}
```

此段代码实现对所创建的 DirectSound 对象的初始化。创建了主缓冲区并设置了 DirectSound 的协调级别。

初始化通知机制，用 WinMain()来初始化捕获缓冲区的通知机制。

```
HRESULT InitNotifications()      //初始化通知机制函数
{
    HRESULT hr;
    if( NULL == g_pDSBCapture )
        return E_FAIL;
    //当声音停止播放时，创建一个通知事件
    if( FAILED( hr = g_pDSBCapture->QueryInterface( IID_IDirectSoundNotify,
                     (VOID**)&g_pDSNotify ) ) )   //获取DirectSound通知接口
        return DXTRACE_ERR_MSGBOX( TEXT("QueryInterface"), hr );
    //创建通知位置
```

```
    for( INT i = 0; i < NUM_REC_NOTIFICATIONS; i++ )
    {
        g_aPosNotify[i].dwOffset = (g_dwNotifySize * i) + g_dwNotifySize - 1;
        g_aPosNotify[i].hEventNotify = g_hNotificationEvent;
    }
    //当在WinMain()函数中初始化捕获缓冲区，则通知DirectSound
    if(FAILED(hr= pDSNotify->SetNotificationPositions( NUM_REC_NOTIFICATIONS,
                                                       g_aPosNotify ) ) )
        return DXTRACE_ERR_MSGBOX( TEXT("SetNotificationPositions"), hr );
    return S_OK;
}
```

此代码主要是初始化通知机制，为了方便录制语音文件。首先通过 QueryInterface()函数获取通知接口，创建通知事件及事件位置，初始化完成后，通知 DirectSound。

初始化格式对话框。

```
HRESULT OnInitFormatsDialog( HWND hDlg )
{
    HRESULT hr;
    if( FAILED( hr = ScanAvailableInputFormats() ) )  //查看可用的声音输入格式
         return DXTRACE_ERR_MSGBOX( TEXT("ScanAvailableInputFormats"), hr );
    HWND hInputList = GetDlgItem( hDlg, IDC_FORMATS_INPUT_LISTBOX );
    FillFormatListBox( hInputList, g_abInputFormatSupported ); //功能实现
    return S_OK;
}
```

此段代码实现对输入格式对话框进行初始化，首先查看用户输入的声音格式，初始化列表并获取对话框句柄，最终实现填充列表框。

设置主缓冲区为一个指定的格式。

```
HRESULT CSoundManager::SetPrimaryBufferFormat(
DWORD dwPrimaryChannels,                      //设置主缓冲区的通道数
DWORD dwPrimaryFreq,                          //设置主缓冲区频率
DWORD dwPrimaryBitRate )                      //设置主缓冲区位率
{  //设置主缓冲区的格式
    HRESULT  hr;
    LPDIRECTSOUNDBUFFER pDSBPrimary = NULL;   //主缓冲区格式
    if( m_pDS == NULL )
        return CO_E_NOTINITIALIZED;
    //获取主缓冲区格式
    DSBUFFERDESC dsbd;
    ZeroMemory( &dsbd, sizeof(DSBUFFERDESC) );
    dsbd.dwSize = sizeof(DSBUFFERDESC);
    dsbd.dwFlags = DSBCAPS_PRIMARYBUFFER;
    dsbd.dwBufferBytes = 0;                   //缓冲区大小
    dsbd.lpwfxFormat = NULL;
    if( FAILED( hr = m_pDS->CreateSoundBuffer( &dsbd, &pDSBPrimary, NULL ) ) )
        return DXTRACE_ERR( TEXT("CreateSoundBuffer"), hr ); //创建声音缓冲区
```

```
    WAVEFORMATEX wfx;                                              //结构体对象
    ZeroMemory( &wfx, sizeof(WAVEFORMATEX) );
    wfx.wFormatTag = (WORD) WAVE_FORMAT_PCM;                       //格式类型
    wfx.nChannels = (WORD) dwPrimaryChannels;                      //主缓冲区通道数
    wfx.nSamplesPerSec = (DWORD) dwPrimaryFreq;                    //主缓冲区采样率
    wfx.wBitsPerSample = (WORD) dwPrimaryBitRate;                  //采样位数
    wfx.nBlockAlign = (WORD) (wfx.wBitsPerSample / 8 * wfx.nChannels);
                                                                   //块对齐大小
    wfx.nAvgBytesPerSec = (DWORD) (wfx.nSamplesPerSec * wfx.nBlockAlign);
    if( FAILED( hr = pDSBPrimary->SetFormat(&wfx) ) )              //设置格式
        return DXTRACE_ERR( TEXT("SetFormat"), hr );
    SAFE_RELEASE( pDSBPrimary );                                   //删除主缓冲区
    return S_OK;
}
```

此段代码实现了对主缓冲区格式的设置。缓冲区中设置了通道数、采样率、采样位数、格式类型等参数。

创建捕获缓冲区并设置格式。

```
HRESULT CreateCaptureBuffer( WAVEFORMATEX* pwfxInput )
{   HRESULT hr;
    DSCBUFFERDESC dscbd;                               //创建DSCBUFFERDESC结构体对象
    SAFE_RELEASE( g_pDSNotify );
    SAFE_RELEASE( g_pDSBCapture );
    //设置通知大小
    g_dwNotifySize = MAX( 1024, pwfxInput->nAvgBytesPerSec / 8 );
    g_dwNotifySize -= g_dwNotifySize % pwfxInput->nBlockAlign;
    //设置缓冲区大小
    g_dwCaptureBufferSize = g_dwNotifySize * NUM_REC_NOTIFICATIONS;
    SAFE_RELEASE( g_pDSNotify );
    SAFE_RELEASE( g_pDSBCapture );                     //删除缓冲区
    //创建捕获缓冲区
    ZeroMemory( &dscbd, sizeof(dscbd) );
    dscbd.dwSize= sizeof(dscbd);
    dscbd.dwBufferBytes = g_dwCaptureBufferSize;       //捕获缓冲区大小
    dscbd.lpwfxFormat= pwfxInput;                      //在创建过程中设置格式
    if( FAILED(hr= g_pDSCapture->CreateCaptureBuffer(&dscbd, //调用捕获缓冲区函数
                                  &g_pDSBCapture, NULL ) ) )
        return DXTRACE_ERR_MSGBOX( TEXT("CreateCaptureBuffer"), hr );
    g_dwNextCaptureOffset = 0;
    if( FAILED( hr = InitNotifications() ) )           //调用通知机制函数
        return DXTRACE_ERR_MSGBOX( TEXT("InitNotifications"), hr );
    return S_OK;
}
```

此代码实现捕获缓冲区的创建。首先要设置好通知大小，再通知应用程序，然后设置创建捕获缓冲区的大小，定义捕获保护区函数创建捕获缓冲区。

创建直接声音缓冲区。

```
HRESULT CSoundManager::Create( CSound** ppSound,
                               LPTSTR strWaveFileName,    //波形文件名
                               DWORD dwCreationFlags,     //创建标志位
                               GUID guid3DAlgorithm,      //3D算法标志
                               DWORD dwNumBuffers )       //缓冲区数量
{
    HRESULT hr;
    HRESULT hrRet = S_OK;
    DWORD   i;
    LPDIRECTSOUNDBUFFER* apDSBuffer = NULL;                //缓冲区指针
    DWORD   dwDSBufferSize = NULL;                         //缓冲区大小
    CWaveFile*  pWaveFile = NULL;                          //声音文件指针
    if( m_pDS == NULL )
        return CO_E_NOTINITIALIZED;
    if( strWaveFileName == NULL || ppSound == NULL || dwNumBuffers < 1 )
        return E_INVALIDARG;
    apDSBuffer = new LPDIRECTSOUNDBUFFER[dwNumBuffers];    //申请缓冲区数组
    if( apDSBuffer == NULL )
    {
        hr = E_OUTOFMEMORY;
        goto LFail;
    }
    pWaveFile = new CWaveFile();                           //开辟声音文件空间
    if( pWaveFile == NULL )
    {
        hr = E_OUTOFMEMORY;
        goto LFail;
    }
    pWaveFile->Open( strWaveFileName, NULL, WAVEFILE_READ ); //打开声音文件
    if( pWaveFile->GetSize() == 0 )
    {
        //如果声音文件是空白的，则不能被创建
        hr = E_FAIL;
        goto LFail;
    }
    //使DirectSound缓冲区与WAV文件相同的大小
    dwDSBufferSize = pWaveFile->GetSize();           //获取大小
    //创建声音缓冲区
    DSBUFFERDESC dsbd;                               //创建结构体对象
    ZeroMemory( &dsbd, sizeof(DSBUFFERDESC) );
    dsbd.dwSize = sizeof(DSBUFFERDESC);              //获取结构体的大小
    dsbd.dwFlags= dwCreationFlags;                   //创建标志位
    dsbd.dwBufferBytes= dwDSBufferSize;              //缓冲区大小
    dsbd.guid3DAlgorithm = guid3DAlgorithm;          //3D算法
    dsbd.lpwfxFormat = pWaveFile->m_pwfx;
```

```
    hr =m_pDS->CreateSoundBuffer( &dsbd, &apDSBuffer[0], NULL );//创建声音缓冲区
    if( hr == DS_NO_VIRTUALIZATION )
        hrRet = DS_NO_VIRTUALIZATION;
    if( FAILED(hr) )
    {
        //如果缓冲区小于DSBSIZE_FX_MIN，则返回DSERR_BUFFERTOOSMALL
      并且通过DSBCAPS_CTRLFX标志位创建缓冲区
            DXTRACE_ERR( TEXT("CreateSoundBuffer"), hr );
       goto LFail;
    }
     //如果再创建一个缓冲区，则通过默认使用DuplicateSoundBuffer()函数复制声音缓冲区。如
果使用DSBCAPS_CTRLFX特性标志位，则通过CreateSoundBuffer()创建缓冲区if( (dwCreationFlags
& DSBCAPS_CTRLFX) == 0 )
    {
        for( i=1; i<dwNumBuffers; i++ )
        {
  if(FAILED(hr=m_pDS->DuplicateSoundBuffer(apDSBuffer[0], &apDSBuffer[i] ) ) )
         {                                //复制声音缓冲区
            DXTRACE_ERR( TEXT("DuplicateSoundBuffer"), hr );
            goto LFail;
         }
      }
    }
    else
    {
      for( i=1; i<dwNumBuffers; i++ )
      {
         hr = m_pDS->CreateSoundBuffer( &dsbd, &apDSBuffer[i], NULL );
         if( FAILED(hr) )               //创建声音缓冲区
         {
              DXTRACE_ERR( TEXT("CreateSoundBuffer"), hr );
              goto LFail;
         }
      }
   }
     *ppSound = new CSound( apDSBuffer, dwDSBufferSize, dwNumBuffers,
                         pWaveFile, dwCreationFlags );  // 动态分配缓冲区空间
      SAFE_DELETE( apDSBuffer );      //删除缓冲区
    return hrRet;
LFail:
    SAFE_DELETE( pWaveFile );         //删除WaveFile声音对象
    SAFE_DELETE( apDSBuffer );        //删除缓冲区
    return hr;
}
```

此段代码实现了对直接声音缓冲区的创建。

输入所选的声音格式。

```
HRESULT OnInputFormatBoxSelected( HWND hDlg )
{
    HWND hInputList = GetDlgItem( hDlg, IDC_FORMATS_INPUT_LISTBOX );
    HWND hOK = GetDlgItem( hDlg, IDOK );  //获得对话框条目句柄
    if( SendMessage( hInputList, LB_GETCURSEL, 0, 0 ) != -1 )
         EnableWindow( hOK, TRUE );         //如果发送消息成功，则使能窗口
    else
         EnableWindow( hOK, FALSE );        //如果发送消息失败，则使能窗口失败
    return S_OK;
}
```

此段代码实现了用户自己所想要录制的声音格式。当选择了所需要的声音格式后，单击“OK”按钮结束。

查看用户的输入格式。

```
HRESULT ScanAvailableInputFormats()
{
    HRESULT  hr;
    WAVEFORMATEX wfx;
    HCURSOR   hCursor;                                   //光标句柄
    DSCBUFFERDESC dscbd;                                 //创建对象
    LPDIRECTSOUNDCAPTUREBUFFER pDSCaptureBuffer = NULL;
         hCursor = GetCursor();                          //获取光标对象
    SetCursor( LoadCursor( NULL, IDC_WAIT ) );           //设置光标
        ZeroMemory( &wfx, sizeof(wfx));
    wfx.wFormatTag = WAVE_FORMAT_PCM;                    //格式类型
    ZeroMemory( &dscbd, sizeof(dscbd) );
    dscbd.dwSize = sizeof(dscbd);                        //结构体大小
    //试着20种不同的标准格式
    for( INT iIndex = 0; iIndex < 20; iIndex++ )
    {
         GetWaveFormatFromIndex( iIndex, &wfx );
         //当输入一个指定的格式，此格式被支持，则创建新的捕获缓冲区
         dscbd.dwBufferBytes = wfx.nAvgBytesPerSec;
         dscbd.lpwfxFormat = &wfx;                              //获取结构体的格式
         if(FAILED(hr=g_pDSCapture->CreateCaptureBuffer(&dscbd,&pDSCapture
         Buffer, NULL ) ) )                                     //创建捕获缓冲区
             g_abInputFormatSupported[ iIndex ] = FALSE;  //不支持输入的格式
         else
            g_abInputFormatSupported[ iIndex ] = TRUE;    //支持输入的格式
         SAFE_RELEASE( pDSCaptureBuffer );
    }
    SetCursor( hCursor );
    return S_OK;
}
```

此段代码实现了对用户所选择的输入格式、拾音设备进行检测，是否支持这 20 种不同标准的格式。

从索引获得波形格式，返回 20 种不同的波形格式。

```
VOID GetWaveFormatFromIndex( INT nIndex, WAVEFORMATEX* pwfx )
{
    INT iSampleRate = nIndex / 4;
    INT iType = nIndex % 4;
    switch( iSampleRate )                                    //选择采样率
    {
        case 0: pwfx->nSamplesPerSec = 48000; break;   //采样率48000Hz
        case 1: pwfx->nSamplesPerSec = 44100; break;   //采样率44100Hz
        case 2: pwfx->nSamplesPerSec = 22050; break;   //采样率22050Hz
        case 3: pwfx->nSamplesPerSec = 11025; break;   //采样率11025Hz
        case 4: pwfx->nSamplesPerSec =  8000; break;   //采样率8000Hz
    }
    switch( iType )                                          //选择采样位数
    {
        case 0: pwfx->wBitsPerSample =  8; pwfx->nChannels = 1; break;//8位单通道
        case 1: pwfx->wBitsPerSample = 16; pwfx->nChannels = 1; break;//16位单通道
        case 2: pwfx->wBitsPerSample =  8; pwfx->nChannels = 2; break;//8位双通道
        case 3: pwfx->wBitsPerSample = 16; pwfx->nChannels = 2; break;//16位
双通道
    }
    pwfx->nBlockAlign = pwfx->nChannels * ( pwfx->wBitsPerSample / 8 );
    pwfx->nAvgBytesPerSec = pwfx->nBlockAlign * pwfx->nSamplesPerSec;
}
```

此段代码的功能是当用户选择要采样的频率和采样位数后，系统对这些格式进行识别并转换。

填充基于可利用的格式列表，具体实现的代码如下。

```
HRESULT FillFormatListBox( HWND hListBox, BOOL* aFormatSupported )
{
    TCHAR strFormatName[255];
    WAVEFORMATEX wfx;
    DWORD dwStringIndex;
    SendMessage( hListBox, LB_RESETCONTENT, 0, 0 );
    for( INT iIndex = 0; iIndex < 20; iIndex++ )
    {
        if( aFormatSupported[ iIndex ] )
        {
            //把索引传递到WAVEFORMATEX结构体中，变为字符串后送入列表中
            GetWaveFormatFromIndex( iIndex, &wfx );      //从索引号获取声音格式
            ConvertWaveFormatToString( &wfx, strFormatName );
                                                          //转换波形文件为字符串
            dwStringIndex = (DWORD)SendMessage( hListBox, LB_ADDSTRING, 0,
```

```
                                        (LPARAM)(LPCTSTR) strFormatName );
                SendMessage( hListBox, LB_SETITEMDATA, dwStringIndex, iIndex );
            }
        }
        return S_OK;
    }
```

此段代码实现了将索引送至列表框，可实现程序对所选输入声音格式的识别。

转换波形文件为文本字符串。

```
VOID ConvertWaveFormatToString( WAVEFORMATEX* pwfx, TCHAR* strFormatName )
{
    wsprintf( strFormatName,
              TEXT("%u Hz, %u-bit %s"),
              pwfx->nSamplesPerSec,    //输出的采样率
              pwfx->wBitsPerSample,    //输出的采样位数
              ( pwfx->nChannels == 1 ) ? TEXT("Mono") : TEXT("Stereo") );
                                       //输出通道数
}
```

此段代码实现将输入缓冲区的数据转换为字符串，并且链接多个字符串和数字。

枚举声音设备。

```
INT_PTR CALLBACK DSoundEnumCallback( GUID* pGUID, LPSTR strDesc,
LPSTR strDrvName, VOID* pContext )
{
    //为20个音频驱动器设置存储空间
    static GUID  AudioDriverGUIDs[20];//定义音频驱动器
    static DWORD dwAudioDriverIndex = 0;
    GUID* pTemp = NULL;
    if( pGUID )
    {
        if( dwAudioDriverIndex >= 20 )
            return TRUE;
        pTemp = &AudioDriverGUIDs[dwAudioDriverIndex++];
        memcpy( pTemp, pGUID, sizeof(GUID) );
    }
    HWND hSoundDeviceCombo = (HWND)pContext;
    //添加字符串到组合框
    SendMessage( hSoundDeviceCombo, CB_ADDSTRING,
            0, (LPARAM) (LPCTSTR) strDesc );
    //获取组合框中的字符串索引
    INT nIndex = (INT)SendMessage( hSoundDeviceCombo, CB_FINDSTRING,
                                  0, (LPARAM) (LPCTSTR) strDesc );
    //存储数据到指针所指的音频设备
    SendMessage( hSoundDeviceCombo, CB_SETITEMDATA,
                 nIndex, (LPARAM) pTemp );
    return TRUE;
}
```

此段代码实现枚举设备对象的功能，通过设定回调函数来枚举设备。

开始或停止缓冲区录音。

```
HRESULT StartOrStopRecord( BOOL bStartRecording )
{
    HRESULT hr;
    if( bStartRecording )
    {               //创建捕获缓冲区并通知捕获缓冲区开始记录
        if( FAILED( hr = CreateCaptureBuffer( &g_wfxInput ) ) )
                return DXTRACE_ERR_MSGBOX( TEXT("CreateCaptureBuffer"), hr );
        if( FAILED( hr = g_pDSBCapture->Start( DSCBSTART_LOOPING ) ) )//开始录制
                return DXTRACE_ERR_MSGBOX( TEXT("Start"), hr );
    }
    else
    {               //停止捕获并读取不能被捕获的数据数据
        if( NULL == g_pDSBCapture )
            return S_OK;
        //停止捕获并读取不能被捕获的数据数据
        if( FAILED( hr = g_pDSBCapture->Stop() ) )     //停止录音
            return DXTRACE_ERR_MSGBOX( TEXT("Stop"), hr );
        if( FAILED( hr = RecordCapturedData() ) )     //重新录音
            return DXTRACE_ERR_MSGBOX( TEXT("RecordCapturedData"), hr );
        //关闭WAV文件
        SAFE_DELETE( g_pWaveFile );                    //删除对象
    }
    return S_OK;
}
```

如果选择开始录音，则创建捕获缓冲区并录音。如果选择停止录音，则停止捕获，停止录音，最后关闭 **WAV** 文件。

录音捕获数据。

```
HRESULT RecordCapturedData()
{
    HRESULT hr;
    VOID*    pbCaptureData = NULL;     //捕获数据
    DWORD    dwCaptureLength;          //捕获长度
    VOID*    pbCaptureData2 = NULL;    //捕获数据
    DWORD    dwCaptureLength2;         //捕获长度
    UINT     dwDataWrote;
    DWORD    dwReadPos;                //读取的位置
    DWORD    dwCapturePos;             //捕获的位置
    LONG     lLockSize;                //锁存的大小
    if( NULL == g_pDSBCapture )
        return S_FALSE;
    if( NULL == g_pWaveFile )
        return S_FALSE;
```

```
    if(FAILED(hr=g_pDSBCapture->GetCurrentPosition(&dwCapturePos, dwReadPos ) ) )
                                                              //获取当前位置
        return DXTRACE_ERR_MSGBOX( TEXT("GetCurrentPosition"), hr );
    lLockSize = dwReadPos - g_dwNextCaptureOffset;
    if( lLockSize < 0 )
        lLockSize += g_dwCaptureBufferSize;
    //锁存块对齐大小为了在一定范围内写数据
    lLockSize -= (lLockSize % g_dwNotifySize);
    if( lLockSize == 0 )
        return S_FALSE;
    //锁存捕获缓冲区
    if( FAILED( hr = g_pDSBCapture->Lock( g_dwNextCaptureOffset, lLockSize,
&pbCaptureData,&dwCaptureLength,&pbCaptureData2, &dwCaptureLength2, 0L ) ) )
        return DXTRACE_ERR_MSGBOX( TEXT("Lock"), hr );
    //写数据到WAV文件
    if( FAILED( hr = g_pWaveFile->Write( dwCaptureLength,
                                    (BYTE*)pbCaptureData, &dwDataWrote ) ) )
        return DXTRACE_ERR_MSGBOX( TEXT("Write"), hr );
    //独立移动捕获偏移量
    g_dwNextCaptureOffset += dwCaptureLength;
    g_dwNextCaptureOffset %= g_dwCaptureBufferSize;
    if( pbCaptureData2 != NULL )
    {
         //写数据到WAV文件
         if( FAILED( hr = g_pWaveFile->Write( dwCaptureLength2,
                                   (BYTE*)pbCaptureData2, &dwDataWrote ) ) )
             return DXTRACE_ERR_MSGBOX( TEXT("Write"), hr );
        //独立移动捕获偏移量
        g_dwNextCaptureOffset += dwCaptureLength2;
        g_dwNextCaptureOffset %= g_dwCaptureBufferSize;
    }
    //解锁捕获缓冲区
    g_pDSBCapture->Unlock( pbCaptureData,  dwCaptureLength,
                          pbCaptureData2, dwCaptureLength2 );
    return S_OK;
}
```

此段代码主要实现从捕获缓冲区中复制数据到输出缓冲区。包括捕获录音数据、获取当前位置、锁存捕获缓冲区、写数据到 **WAV** 文件、解锁捕获缓冲区这几个阶段。

保存声音文件，当用户需要保存声音文件时调用。

```
VOID OnSaveSoundFile( HWND hDlg )
{
    HRESULT hr;
    static TCHAR strFileName[MAX_PATH] = TEXT("");     //文件名
    static TCHAR strPath[MAX_PATH] = TEXT("");         //文件路径
    //创建OPENFILENAME结构体
```

```
    OPENFILENAME ofn = { sizeof(OPENFILENAME), hDlg, NULL,
                         TEXT("Wave Files\0*.wav\0All Files\0*.*\0\0"), NULL,
                         0, 1, strFileName, MAX_PATH, NULL, 0, strPath,
                         TEXT("Save Sound File"),
                         OFN_OVERWRITEPROMPT | OFN_PATHMUSTEXIST |
                         OFN_HIDEREADONLY | OFN_NOREADONLYRETURN,
                         0, 0, TEXT(".wav"), 0, NULL, NULL };
    //获取默认的媒体路径(如C:\WINDOWS\MEDIA)
    if( '\0' == strPath[0] )
    {
        if( GetWindowsDirectory( strPath, MAX_PATH ) != 0 )  //获取窗口目录
        {
            if( strcmp( &strPath[strlen(strPath)], TEXT("\\") ) )
                strcat( strPath, TEXT("\\") );
            strcat( strPath, TEXT("MEDIA") );
        }
    }
    if( g_bRecording )
    {                                                         //停止捕获
        StartOrStopRecord( FALSE );
        g_bRecording = FALSE;
    }
    //更新用户界面控制来显示所加载的声音
    EnableWindow( GetDlgItem( hDlg, IDC_RECORD ), FALSE );
    SetDlgItemText( hDlg, IDC_FILENAME, TEXT("Saving file...") );
    //显示保持文件名对话框，此时加载指定文件
    if( TRUE != GetSaveFileName( &ofn ) )
    {                                                         //获取保存文件名
        SetDlgItemText( hDlg, IDC_FILENAME, TEXT("Save aborted.") );
        return;
    }
    SetDlgItemText( hDlg, IDC_FILENAME, TEXT("") );   //设置对话框项目文本
    SAFE_DELETE( g_pWaveFile );                           //删除对象
    g_pWaveFile = new CWaveFile;
    if( NULL == g_pWaveFile )
        return;
    WAVEFORMATEX wfxCaptureWaveFormat;                        //得到捕获缓冲区的格式
    ZeroMemory( &wfxCaptureWaveFormat, sizeof(WAVEFORMATEX) );
    g_pDSBCapture->GetFormat(&wfxCaptureWaveFormat,      //获取文件格式
  sizeof(WAVEFORMATEX), NULL );
    //加载WAV声音文件
    if(FAILED(hr=g_pWaveFile->Open(strFileName,&wfxCaptureWaveFormat,
WAVEFILE_WRITE ) ) )                                      //打开WAV文件
    {
        DXTRACE_ERR_MSGBOX( TEXT("Open"), hr );
        SetDlgItemText( hDlg, IDC_FILENAME, TEXT("Can not create wave file.") );
```

```
        return;                                  //若打开文件失败，则显示不能创建WAVE文件
    }
    //更新用户界面控制，显示所加载的声音
    SetDlgItemText( hDlg, IDC_FILENAME, strFileName );     //设置对话框项目文本
    EnableWindow( GetDlgItem( hDlg, IDC_RECORD ), TRUE ); //使能窗口
    //记住下次路径
    strcpy( strPath, strFileName );          //把打开的声音文件路径复制并显示
    char* strLastSlash = strrchr( strPath, '\\' );
}
```

此代码主要实现保存捕获的声音文件。首先创建要保存声音文件的结构体，然后停止捕获，获取捕获缓冲区的格式，加载捕获缓冲区的 WAV 文件，保存声音文件。

6.3.3 程序运行结果

上节详细介绍了具体的实现代码，并已设计完成。接下来就要验证所设计的程序是否正确，下面就对程序进行编译和调试。

1. 编译和运行程序

首先选择“生成”→“编译”命令，或者按“Ctrl+F7”组合键，生成解决方案并生成可指向文件。如果在界面的下方窗口中产生错误信息，用户就可以在错误信息行双击鼠标，立刻会自动把错误定位在产生错误的代码行，随即可以对其修改。并重复上述操作，直至没有错误，编译成功为止。

选择“调试”→“开始运行（不调试）”选项，或者按下“Ctrl+F5”组合键，运行和执行该程序。具体的调试方法可见第 3 章。

2. 运行程序

当编译调试成功后，即可使用本案例，下面具体讲解一下本案例。

STEP 01 启动本案例后，首先弹出一个对话框，来选择捕获声音的设备，可以在下拉列表中选择所需要的捕获设备。选定后，单击“OK”按钮，进入下一步，如图 6-4 所示。

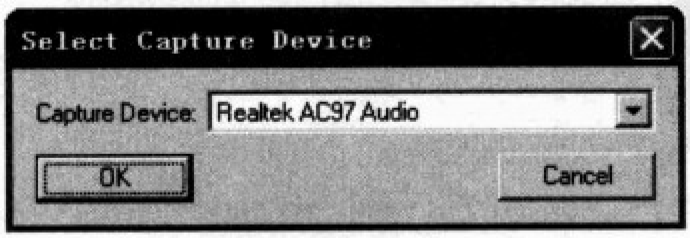


图 6-4　选择捕获声音设备

STEP 02 当选择好了捕获声音设备后，就会弹出一个对话框，让用户选择捕获声音的格式，此格式设置了采样率和采样位数。

在捕获录制声音时一般选择的格式为 22050Hz、16bit，即采样率为 22050Hz，采样位数为 16bit。在这里选择了 48000Hz、8bit 的捕获格式，然后单击“OK”按钮进入到选择捕获声音对话框，如图 6-5 所示。

STEP 03 单击“Sound file”按钮，选择要录制的声音，在状态文本框中会显示刚才所选的声音文件的路径，在对话框下方同时也会显示上面所选的输入格式。单击“Record”按钮对所选声音进行录制，如图 6-6 所示。

图 6-5　选择录音格式

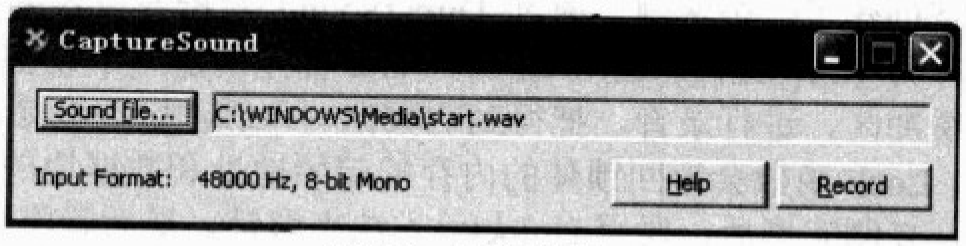


图 6-6　录制声音

6.4　DirectSound录制语音文件开发常见问题

在利用 DirectSound 录制自己的 WAVE 音频文件时会遇到一些问题，下面就列举了几个在开发过程中常见的问题及解决方法。

- **用 DirectSound 制作录音程序时，如何能实时得到输入音频设备的声音强度？** MixerGetContrlDetails 等函数可以提供一些帮助，MixXXX 系列的 API 函数可以监控声卡的输入和输出的系统音量。MixerGetContrlDetails 可以得到与一个音频线程相联系的单个控制信息，如扬声器的音量等。首先打开 Mixer 设备与声卡联系起来，得到要监控的线程的 Mixer 控制，然后获得控制的信息。
- **如何获得捕获缓冲区的属性信息？** 捕获缓冲区的属性可以从 CaptureBuffer.Caps 属性获得，CaptureBufferCaps 结构体描述缓冲区的大小、是否支持特性等。当缓冲区被创建时，要得到关于缓冲区里的数据格式信息，应使用 CaptureBuffer.Format 属性。要检查捕获缓冲区的状态如何，就检查 CaptureBuffer 属性。如果此缓冲区被捕获，则 Capturing 属性会被设置。Looping 在前面调用 CaptureBuffer.Start 方法时被设置，如果这种情况常常发生，则 Looping 就会被设置。CaptureBuffer.GetCurrentPosition 方法返回在缓冲区中的读指针和写指针的偏移量。
- **如何设置缓冲区的捕获通知？** 当捕获音频时，如何使应用程序被通知何时让读指针到达缓冲区中一个特定的点，或者是缓冲区被停止时刻。通过使用 Notify.SetNotificationPosition 方法，可以设置任何在事件将被通知的缓冲区的点数目。当缓冲区被捕获时，就不能设置了。要建立通知，按下列步骤执行。

STEP 01 为每个通知位置建立一个 AutoResetEvent。

STEP 02 由 CaptureBuffer 对象的 Notify 构造器获得通知对象。

STEP 03 建立一个 BufferPositionNotify 结构体数组，每个成员对应一个通知位置。设置 Offset 成员，这是被通知位置的字节偏移量。设置 EventNotifyHandle 成员到用户第一步所建立的事件之一的 AutoResetEvent.Handle。

STEP 04 调用 Notify.SetNotificationPositions。

然后就可以启动缓冲区的一个独立线程，并使用 WaitHandle.WaitAny 来等待通知。

6.5 本章小结

本章创建了一个案例。通过 DirectSound 对拾音设备采集的声音数据进行捕获，并进行录音保存为 WAV 文件。利用 DirectSound 首先创建一个设备对象，然后利用设备对象创建一个捕获缓冲区，进行录音，把得到的数据写入缓冲区，在缓冲区中设置通知机制。然后锁住缓冲区，Lock 函数会返回锁住的内存的起始地址和数据块的大小。其次就是要从缓冲区中复制数据，复制完毕后，要用 Unlock()方法来解锁。最后就将缓冲区的数据保存为 WAVE 文件格式。在录制语音文件时，可能会遇到一些棘手的问题，在本章最后介绍了开发中常见的问题和解决方法。

第3篇

视频开发

本篇详细讲解了微软 DirectShow SDK 和 VFW 的视频开发技术。介绍了 DirectShow SDK 的获取、安装、Visual C++ 2005 开发环境配置和常见开发问题，以及 DirectShow 的工作原理、滤波器（Filters）构成和分类、滤波器链表工作原理、事件通知机制、常用的 DirectShow SDK 接口等。使用 DirectShow SDK 可以快速构建播放多媒体文件、捕获采集音频/视频的应用程序。该部分以两个案例分析 DirectShow SDK 的开发流媒体回放、音/视频采集过程。同时还介绍了 VFW 的视频采集功能，以案例下的两个应用 VFW 的压缩管理器（ICM）和用户算法分别实现视频编码，阐述视频通用平台。通过本篇的学习，读者可以掌握视频开发的工具平台，基于本部分的案例设计出功能更丰富的视频应用程序。

- 第 7 章　DirectShow 开发基础
- 第 8 章　打造自己的媒体播放器
- 第 9 章　自制 DV 实现视频采集
- 第 10 章　VFW 技术实现视频处理通用平台

[illegible]

[illegible] 人 [illegible]

第 7 章

DirectShow 开发基础

相信对基于 VC＋＋开发视频的研发人员来讲，DirectShow SDK 这个概念应该不会陌生，或者有所耳闻。它是 Microsoft 微软公司为简化或降低音/视频开发的难度而为音/视频应用开发人员定制的开发包、软件库。它是我们熟悉的 DirectX 家族中的一个组件，主要针对程序员开发数字音/视频程序提供支持。使用 DirectShow 我们可以在很短的时间内搭建自己的视频播放、捕获和编辑应用程序，不必对有关视频硬件如显卡、视频捕获设备、视频编码算法、解码算法，音/视频数据同步等，复杂的技术做细节的掌握。另外，由于现在的视频捕获设备如USB摄像头、视频捕获卡、显卡等主流的驱动均是WDM模式，VC++开发人员自己去访问、控制、读取此类设备的数据显然不是件容易的事情。DirectShow 却可以很灵活地帮助我们去完成工作，即屏蔽了我们直接对硬件的操作，既降低了开发难度，又可快速开发出我们自己的产品。

本章重点

- Microsoft DirectShow SDK
- 开发环境配置
- DirectShow SDK 基本架构
- DirectShow SDK 常用开发案例

7.1 Microsoft DirectShow SDK

DirectShow SDK 是 DirectX 中的一个组件，而 DirectX 是微软公司为流媒体应用开发的一套 API 函数。使用 DirectShow SDK 可以快速构建流媒体的采集和影音文件的回放应用程序，并且 DirectShow SDK 是免费的，微软官方网站提供下载。

7.1.1 DirectX及成员组成

在讲 DirectShow 之前，首先我们不得不介绍一下 DirectX（简称为 DX）。DirectX 是微软推出的一套基于 Windows 操作系统的流媒体应用程序接口函数 API。DirectX 在开发中可分为两个部分，一个是运行库（Run Time Library），使用 DirectX 编译出来的程序必须要有运行库的支持；另外一个是开发库，也就是常说的 SDK（软件开发包 Soft Develop Kits），它是在编译 DirectX 程序中所必需的。

DirectX 是一种图形应用程序 GUI 接口 API、多媒体处理加速接口 API，简单地说它是微软为提高系统性能、加速软件开发的辅助软件。安装 DirectX 后，操作系统中的软件会更直接地利用硬件加速资源，从而提高访问硬件的速度。

在 Windows 操作系统的体系构架中，在 OS 内核与硬件之间有一层抽象层 HAL（Hardware Abstract Layer），专门对硬件进行屏蔽和抽象，所以用户也就不再被允许对硬件进行直接的访问。这样做可以大大提高操作系统的抗破坏性和抗干扰性，但同时，硬件操作的效率也会随之降低，许多新硬件的新特性也无法直接使用，这对海量、实时多媒体和游戏的应用显然是不利的。而 DirectX 就是微软公司提供的一整套优秀的应用程序编程接口（API）、SDK，用于联系应用程序和硬件本身，这对 Windows 平台下的多媒体应用程序和电脑游戏的发展起到了关键性的作用。

DirectX 组件包括 DirectGraphics、DirectInput、DirectPlay、DirectSetup、DirectMusic、DirectSound、DirectShow、DirectX Media Objects 等。

- DirectX Graphics：集成了以前的 DirectDraw 和 Direct3D 技术。
- DirectDraw：主要负责 2D 加速，以实现对显卡内存和系统内存的直接操作。
- Direct3D：主要提供三维绘图硬件接口，它是开发三维 DirectX 游戏的基础。
- DirectInput：支持输入服务（包括鼠标、键盘、游戏杆等），同时支持输出设备。
- DirectPlay：提供多人网络游戏的通信、组织功能。
- DirectSetup：提供自动安装 DirectX 组件的 API 功能。
- DirectMusic：支持音频的捕获、回放功能。
- DirectSound：提供音频捕捉、回放、音效处理、硬件加速和直接设备访问等功能。
- DirectShow：在 Windows 平台上处理各种格式的媒体文件的回放、音/视频采集、非线性编辑等高性能要求的多媒体应用，提供了完整的解决方案。

- DirectX Media Objects：DirectShow Filter 的简化模型，提供更方便的媒体流数据处理方案。

7.1.2 DirectShow SDK能做什么

DirectShow SDK 从 VC++开发人员角度来看是 SDK（Software Development Kits，软件开发包），更直接点讲就是静态（LIB）、动态链接库（DLL）。从计算机软件专业角度来讲是 COM 组件，使用 DirectShow 开发的应用程序是 COM 组件的客户程序。一般的 DirectShow 应用程序如捕获、播放视频不需要对 COM 了解过深，但开发自己的 Filter 组件就要需要对 COM 有较为细致的掌握，由于本书是案例开发和分析，Filter 的具体开发细节见 DirectShow 的在线帮助。

Microsoft DirectShow SDK 是为多媒体数据流的应用而开发的。它支持多媒体流的捕获（Capture）和预览（Preview）。运用 DirectShow，用户可以很方便地从支持 WDM 驱动模型的采集卡上捕获数据，并且可以进行相应的后期处理甚至存储到文件中；同时可以方便地构建影音文件的回放、编辑等操作。使用 DirectShow 能够完成以下工作。

- 音/视频多媒体流的捕获和预览。
- 支持多种媒体格式 ASF、MPEG、AVI、MP3、WAVE 的回放。
- 集成其他 DirectX 技术，增强音/视频硬件如声卡、显卡的性能。
- 视频文件的回放、非线性编辑等。
- 支持 DVD、DV 等设备。
- 定制自己的 Filter。

7.1.3 如何获取DirectShow SDK

DirectShow SDK 最初并不是 DirectX 的成员，是从 DirectX 6.0 才出现的。目前主流使用的是 DirectX SDK 8.0 和 DirectX SDK 9.0，而到 DirectX SDK 9.0 以后的版本 DirectShow 又被移出了 DirectX，被包含在了 Windows SDK 中。另外特别需要提醒用户的是，DirectX 9.0 SDK 也有几个版本，其中 DirectX 9.0C SDK 并不包含 DirectShow SDK，而 DirectX 9.0B SDK 则包含 DirectShow SDK。

所以用户在下载较新的 DirectX SDK 时，安装后如果没有 DirectShow SDK，则需要下载 DirectX SDK Extras 之类的升级包。例如 Visual Studio 2005 Professional Edition（VS2005）中包含了 DirectX SDK，但并没有 DirectShow SDK，则用户需要再下载 DirectX SDK Extras 升级包。其实用户可以直接从微软官方网站下载 DirectX 9.0 Complete Software Development Kit（SDK），文件名为 dx9sdk.exe。不过下载时要默认安装 Windows 正版软件验证插件。

作者开发用的 DirectShow SDK 是从某网站外挂链接的 dx90bsdk.exe，即 DirectX 9.0B SDK。并且 dx90bsdk.exe 是 2003 年发布的，而 dx9sdk.exe 是 2002 年发布的。官方网址为 http://download.microsoft.com/download/b/6/a/b6ab32f3-39e8-4096-9445-d38e6675de85/dx90bsdk.exe，且作者测试过，用下载软件（如迅雷）可以绕过正版验证的问题！

总地来说，读者要清楚 DirectShow SDK 的演变过程，清楚自己需要的 SDK 是否安装完整。另外，也可以到网站上利用搜索引擎查找感兴趣的资料或下载软件安装包。相信在网络时代的今天，我们肯定能够搜集到用户常用的软件包，比如 DirectShow SDK。有趣的是用户常用的 VC++ 2005（作者也曾使用），即 Visual Studio 2005 Professional Edition 英文版，就是从网站上下载的（2.69GB），市面上买到的绝大部分都是中文版的，当然中文版也可以从网络上轻松下载。

7.1.4 安装DireShow SDK

有了 DirectShow SDK 软件安装包，下面就让我们开始安装它吧。因为从网络上下载的一般是*.exe 可执行压缩安装包，需要先解压再安装。假定已经安装的操作系统为 Windows XP SP2。如 dx90bsdk.exe 解压到当前目录，假设我们已经有安装包文件夹\dx9bsdk，则双击运行文件里面的 Install.exe 可执行程序，运行结果如图 7-1 所示。

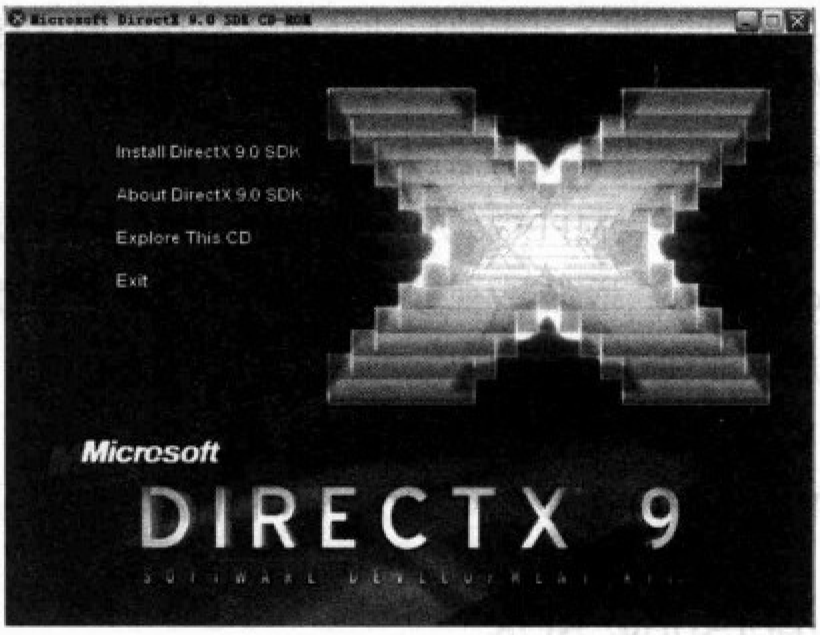


图 7-1　DirectX 9.0 SDK

STEP 01　单击运行图 7-1 中的“Install DirectX 9.0 SDK”按钮，得到如图 7-2 所示的画面。

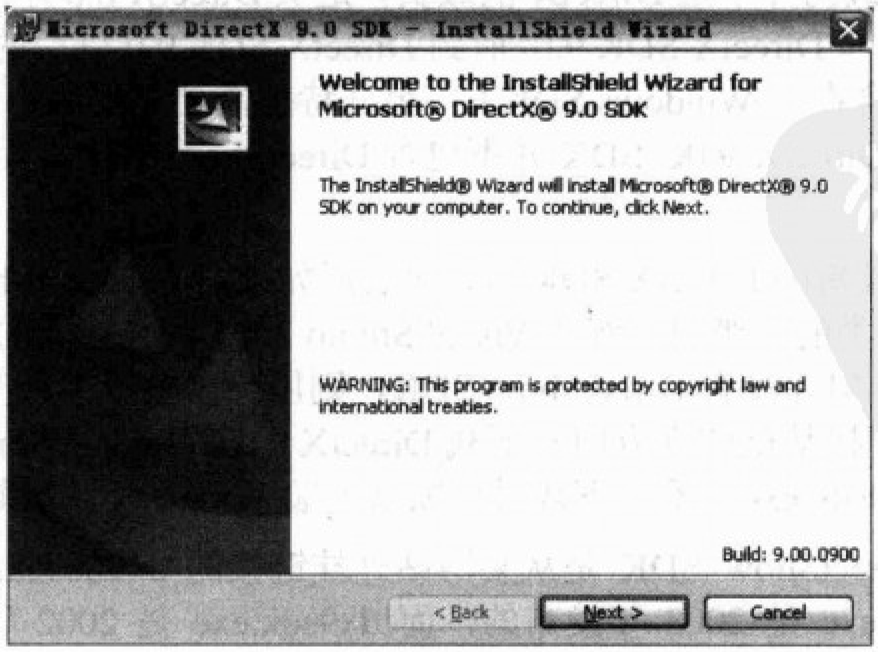


图 7-2　DirectX 9.0 SDK 安装向导

STEP 02 单击“Next”按钮，在弹出的对话框中同意许可条款 ⊙I accept the terms in the license agreement ，如图 7-3 所示。

STEP 03 定制要安装的内容，如图 7-4 所示。

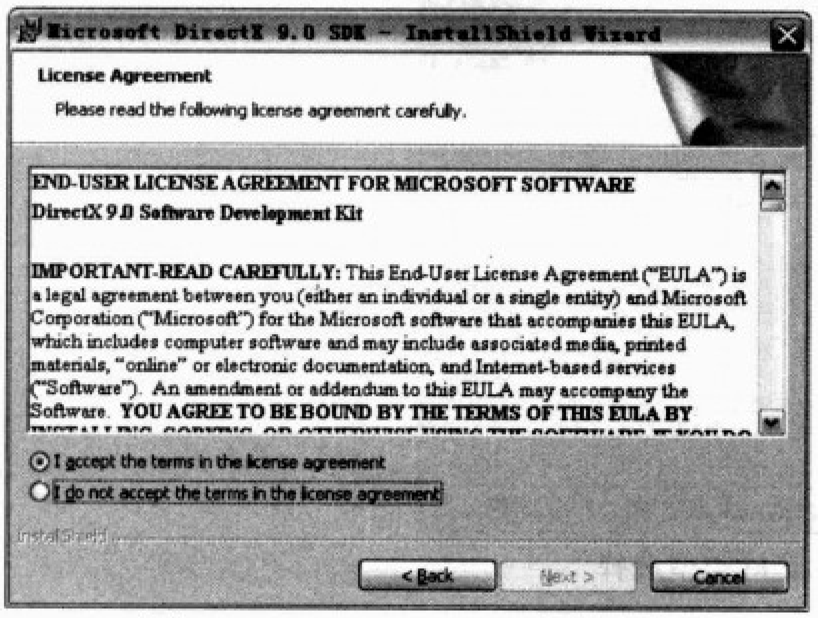


图 7-3 DirectX 9.0 SDK 安装向导

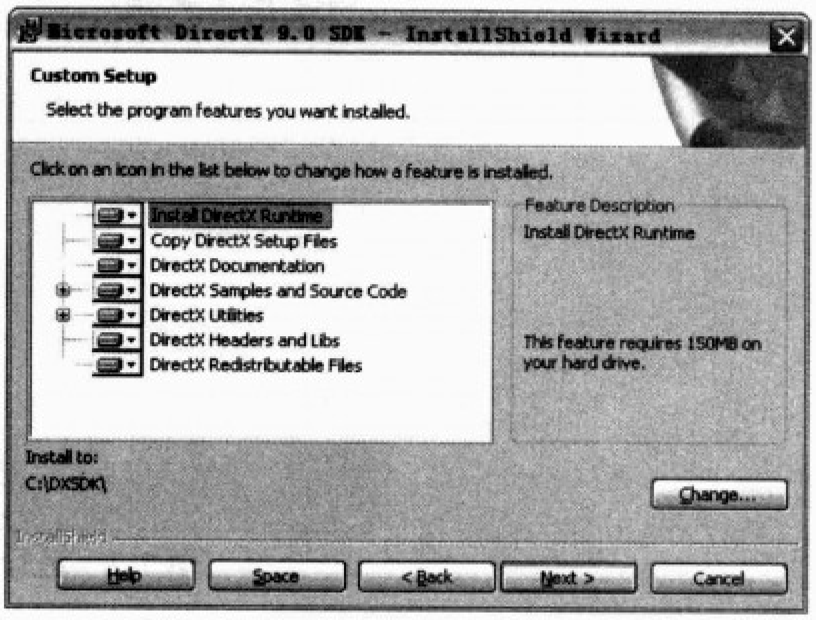


图 7-4 DirectX 9.0 SDK 安装向导

STEP 04 这里只安装 C++的例程源代码，如图 7-5 所示。

STEP 05 单击“Change”按钮，更改安装目录为“D:\DXSDK”，如图 7-6 所示。考虑到大多数的 C 盘为系统盘，为不影响操作系统的运行效率，强烈建议用户把 DXSDK 安装到非系统盘，作者安装到 D 盘。

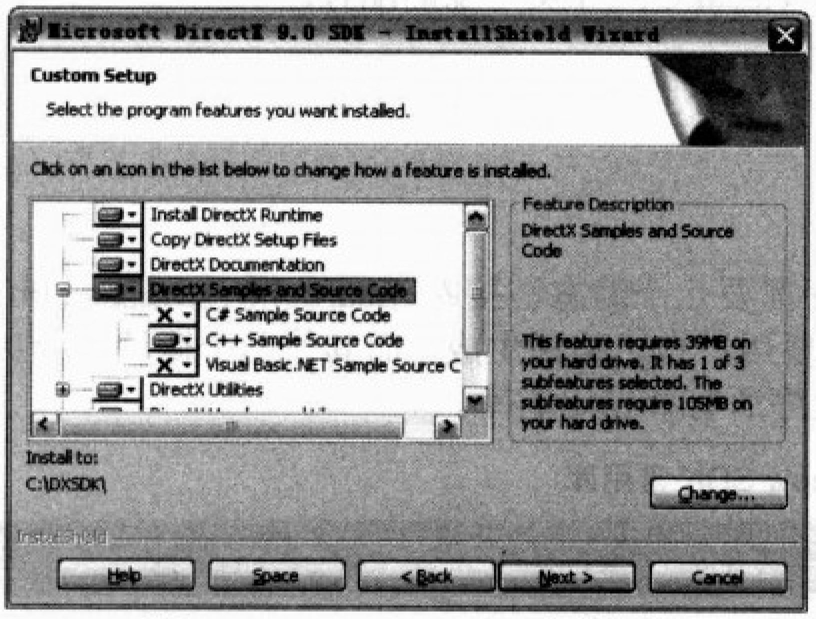


图 7-5 DirectX 9.0 SDK 安装向导

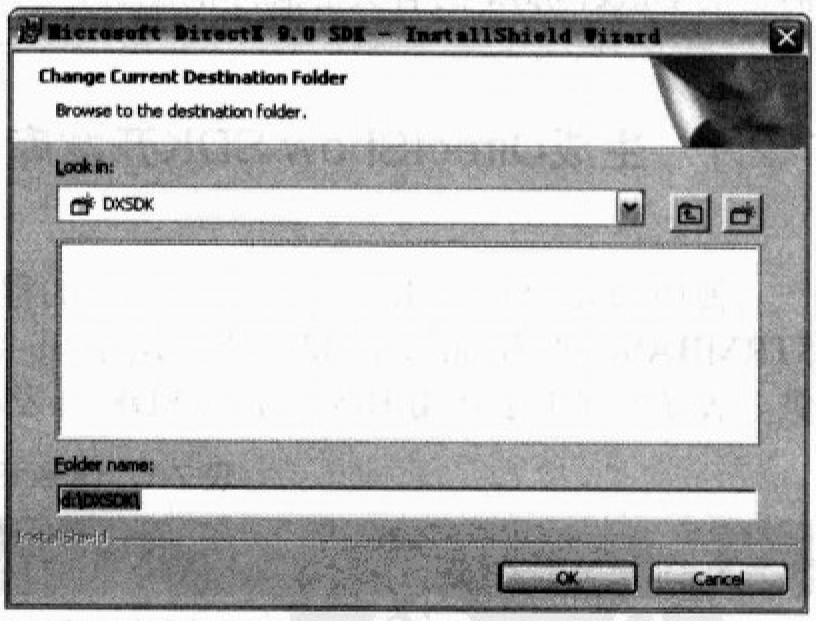


图 7-6 DirectX 9.0 SDK 安装向导

STEP 06 单击“OK”按钮，接着单击“Next”按钮，出现如图 7-7 所示的画面。

STEP 07 如图 7-7 所示，Retail 运行支持：将安装非 DirectX 9.0 组件。Debug 运行支持：\retail 文件夹包含安装程序，它将安装非 DirectX 8.0 产品部件，\debug 文件夹安装所有排错部件。

STEP 08 单击图 7-7 中的“Install Now”按钮，安装程序开始安装用户的定制功能。

安装完 DirectX SDK，用户可以先概览其相关内容：升级内容、安装、新特征，以及每部分的最新变化，双击 dxreadme.htm 文件。

按照上述步骤安装好软件之后就可以开始使用 DirectX SDK 来开发用户自己的程序了。

但是在开发自己的程序之前，首先需要配置 Visual C++开发环境和编译生成 DirectShow SDK 开发库。

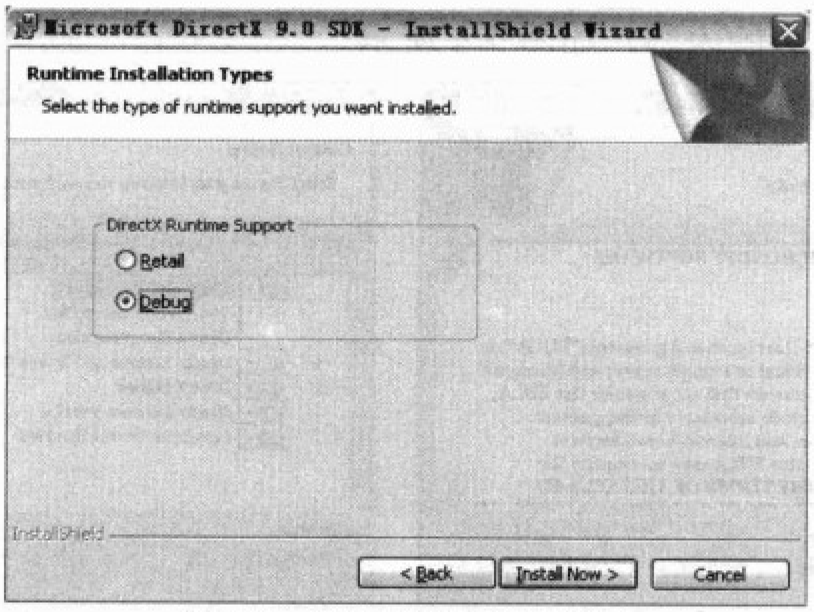


图 7-7　运行库支持类型选择

7.2　开发环境配置

开发环境的配置主要有两个工作要做：一是在使用 DirectShow SDK 开发自己的程序时需要的 DirectShow 的有关静态库的配置，二是 Visual C++开发环境的配置。

7.2.1　生成DirectShow SDK开发库

使用 DirectShow SDK 开发用户自己的程序需要几个静态链接库：quartz.lib、strmbasd.lib、STRMBASE.lib 和 strmiids.lib。中间两个 lib 需要用户自己编译生成，而其他两个微软已经提供。表 7-1 列出了使用 DirectShow SDK 开发程序所有要使用的库。

表 7-1　DirectShow SDK 常用库

库名	功能说明
strmiids.lib	定义了 DirectShow 标准的输出类标识（CLSID）和接口标识（IID）
strmbasd.lib	流媒体开发用到的库，Debug、Debug_Unicode 版本
strmbase.lib	流媒体开发用到的库，Release、Release_Unicode 版本
quartz.lib	定义了导出函数 AMGetErrorText
winmm.lib	使用 Windows 多媒体编程用到的库

基于 VC++ 2005 开发软件使用 DirectShow SDK，首先需要用户编译 DirectShow 自带的源代码工程 baseclasses，以生成 DirectShow SDK 的不同版本的库。同时由于 DirectShow SDK 是早期的 VC 开发软件，所以使用 VC++ 2005 编译 DirectShow SDK 会出现很多编译问题。下面就列出了详细的编译过程和问题分析、解决方法。

1. 启动 VC++ 2005

启动 VC++ 2005 可以采用以下 3 种方式中的一种。

- 运行“开始”→“程序”→“Microsoft Visual Studio 2005”→“Microsoft Visual Studio 2005”命令。
- 制作快捷方式到桌面以方便快捷使用。
- 选择“开始”→“运行”命令，在弹出的对话框中输入“devenv（develop environment，开发环境）”，快速启动 VC++ 2005。

为创建 strmbasd.lib、strmbase.lib 静态库，选择“文件”→“打开”→“项目/解决方案”命令，在弹出的对话框中打开“BaseClasses”项目，路径如图 7-8 所示。

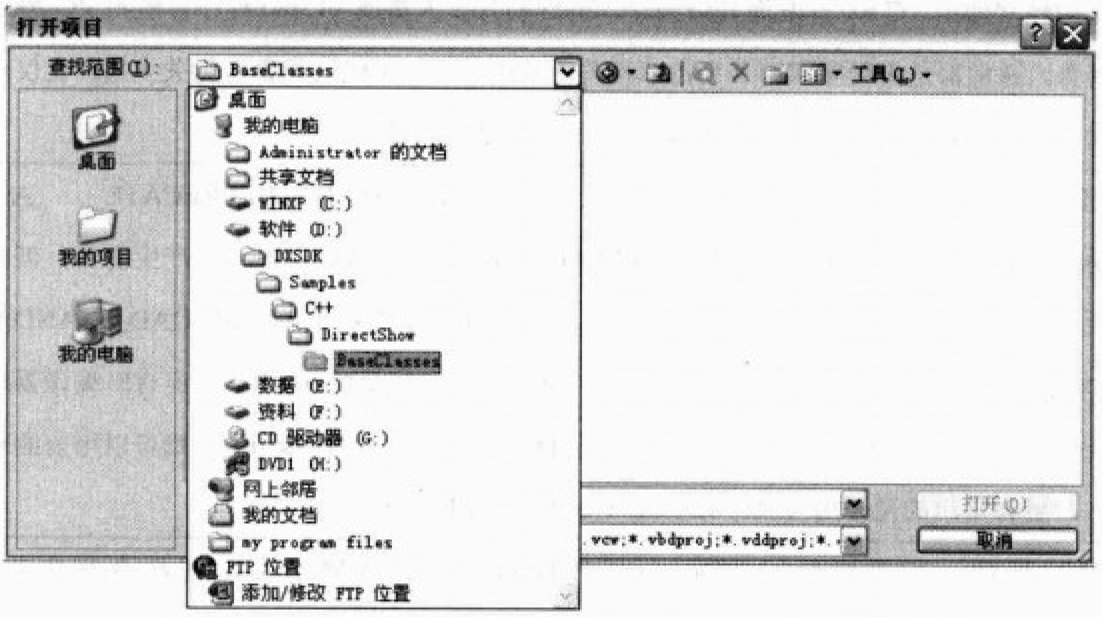


图 7-8　BaseClasses 项目路径

打开“baseclasses.sln”项目。如果 VC 2005 有提问，则默认同意或确定。现在就开始编译该项目。按“F7”快捷键可以编译生成项目。初次编译 VC 2005 会报很多错误或者警告，有的需要我们手工修改程序，或者修改 VC 2005 环境配置或编译选项；有的是一类问题，解决方法也有多种。下面列出了编译过程中所有出现的错误或警告，首先定位问题，然后分析出现问题的原因，最后解决问题。

2. 常见问题解析

由于 DirectShow SDK 是 2003 或 2002 年的版本，在 VC++ 2005 平台上编译的时候会出现很多问题，现就编译过程遇到的常见问题作定位和解析。

1）error C2146

语法错误：缺少“;”（在标识符“PVOID64”的前面）。

问题定位	头文件 winnt.h 中（222 行），typedef void * POINTER_64 PVOID64;
问题分析	编译器通知我们 POINTER_64 没有定义。搜索后，发现定义在 VC2005 安装目录下的 baseTsd.h 文件中。可能由于某种原因那部分代码没有被执行到
问题解决	在错行之前添加#define POINTER_64 __ptr64

2）error C4430

缺少类型说明符，假定为 int。注意：C++不支持默认 int。

问题定位	头文件 ctlutil.h 中（278 行），operator=(LONG);
问题分析	函数定义中，在以前 VC 版本中，如果没有显示的定义变量类型，则编译器将其视为默认整型；但是 VC2005 不支持默认整型
问题解决	① 用 wd4430（Warning Define 4430，即定义的 Warning 级别）来解决；途径：项目→属性→配置属性→C/C++→命令行→附加选项，添加/wd4430。② 手工修改，指定类型：LONG operator = (LONG);

3）warning C4996

问题定位	\wxdebug.cpp（534 行）：warning C4996："_vsnprintf"被声明为否决的
问题解析	这是由经常使用 VC 6.0 开发而升级到 VC 2005 的开发人员或 VC 6.0 程序移植到 VC 2005 平台时遇到的一大类警告，通常都是由程序调用了字符串相关函数引起的。虽然这类警告无关大局，仅仅意味着使用的函数已过时（deprecated）或不安全，编译器要求用新的函数来替代
问题解决	使用 _CRT_SECURE_NO_DEPRECATE、_SCL_SECURE_NO_DEPRECATE、_ATL_ SECURE_NO_DEPRECATE、_AFX_SECURE_NO_DEPRECA 等宏来禁止 VC2005 对此产生警告，但仍然使用非安全的库函数。然而更好的解决办法是定义一个宏：_CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES，那么 VC 将会自动替换并使用相关的安全库 Safe Library 来代替 C/C++标准库。但有时编译器不能自动替换对应的安全库，而我们自己的程序又确认是没有逻辑错误而只是由编译器引起，则可以用前面对应的宏来禁止此类警告，更为常用的是使用_CRT_SECURE_NO_DEPRECATE。
解决途径	项目→属性→配置属性→C/C++→预处理器→预处理器定义，在命令行中添加_CRT_SECURE_NO_DEPRECATE

另外，在编程时为避免出现此类问题，微软提出 10 点注意事项，详见本书的"附录 A 旧版 VC 升级到 VC 2005 程序安全的 10 点注意事项"。

4）error C4430

缺少类型说明符，假定为 int。

注意：

C++ 不支持默认 int。

问题定位	.\wxdebug.cpp（567 行）。static g_dwLastRefresh = 0;
问题分析	此问题同问题 2）。其实这也是一大类问题，VC 2005 不支持默认整型。需要显式定义所有变量的类型，尽管是常用的整型
问题解决	显式定义。 static DWORD g_dwLastRefresh = 0;

5）error C2065

Count：未声明的标识符。

问题定位	.\winutil.cpp（2104 行）：for (Count = 0;Count < Result;Count++) {
问题分析	这类问题同样是由于旧版 VC 程序编译规则同 VC 2005 的区别，Count 是在循环体内定义的变量，只能在循环内使用。出了循环之后"生命"就结束了，所以将它定义到外面去。其他类似错误 error C2065，均为变量声明的位置错误，因为声明的变量在出循环之外无效了
问题解决	为所有使用该变量的地方有效，把变量定义移到循环体之外。UINT Count = 0;移动到函数入口的开始或循环体以外，同时循环中的变量定义取消

6）error C4430

缺少类型说明符，假定为 int。

问题定位	\wxdebug.cpp（564 行）：static g_dwLastRefresh = 0;
问题解决	问题同 4），显式定义变量类型：static DWORD g_dwLastRefresh = 0;

7）error C2065

iDone：未声明的标识符。

问题定位	\outputq.cpp（664 行）：*nSamplesProcessed = iDone - iLost;
问题解决	问题同 5），变量为循环体内声明，循环体之外定义无效。把 iDone 的定义移动到函数入口开始处或者循环体以外，同时把循环体内的变量定义取消

把如下代码：

```
for (long iDone = 0;
         iDone < nSamples || (m_nBatched != 0 && m_bSendAnyway);
```

修改为：

```
LONG iLost = 0;
long iDone = 0;
for (iDone = 0;
         iDone < nSamples || (m_nBatched != 0 && m_bSendAnyway);
```

注意：

针对错误 C2065，假如需要修改的语句太多，则有一种非常快捷的方式：修改 VC 2005 属性配置，取消循环体内的变量一致性。如图 7-9 所示，把"强制 For 循环范围中的一致性"值修改为"否（/Zc:forScope-)"。这样不会因为大量旧版程序中的同一个问题而修改更多的代码，当然开发人员使用 VC2005 开发程序时，也需要养成良好的习惯，变量在函数入口或循环体外定义、声明。

3. 成批编译（Batch Build）生成 4 个版本的静态库

成批编译生成 Debug、Debug_Unicode、Release、Release_Unicode4 个版本的静态库。

用户经过上面的常见问题解析，项目一般都能成功编译。如果出现下面所示的提示，就表示该程序编译成功。

```
========== 生成: 1 已成功, 0 已失败, 0 最新, 0 已跳过==========
```

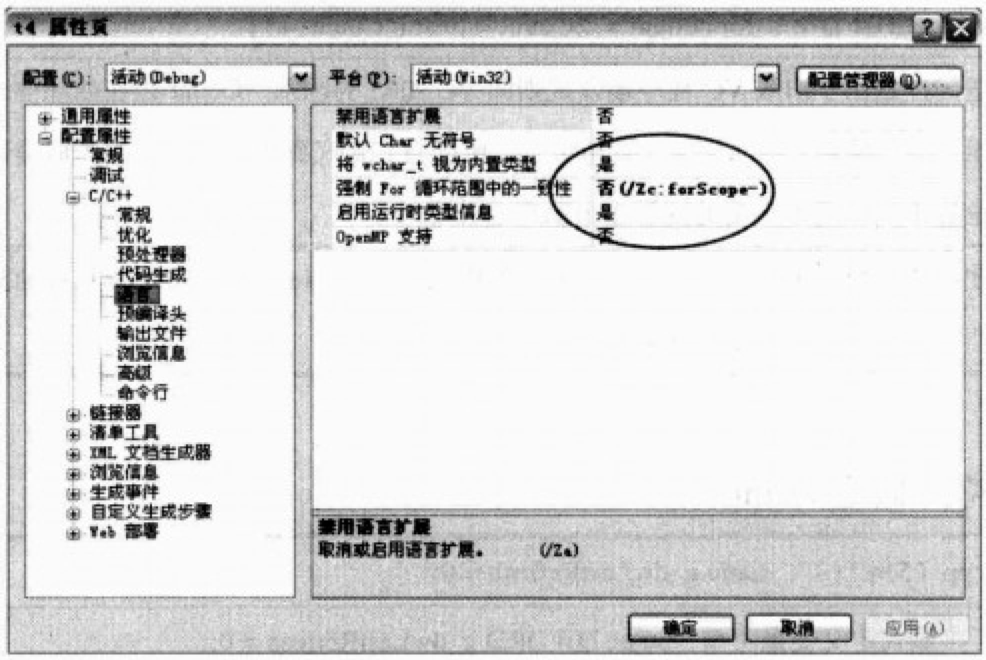


图 7-9 “属性页”对话框

现在编译项目“BaseClasses”的各种不同版本的静态库，以方便用户后续的不同开发使用。菜单：生成\批生成（T），如图 7-10 所示。

注意：

用户在使用 VC 6.0 时，编译生成一般有两个版本，即 Debug 和 Release，而在 VC 2005 中，则又新增了两个版本，即 debug 和 release。因为在 VC 2005 中，在 unicode 模式下，一个字符占用 2 个字节，而旧版 VC 一个字符是占 1 个字节，即是 ANSI 格式。为考虑兼容，VC 2005 提供了 4 种版本的生成，开发人员根据需要使用不同的生成。

ANSI 字符表示是 char，占 8 位；Unicode 字符表示是 wchar_t，占 16 位。Windows2000 以后的操作系统都是基于 Unicode 开发的平台，而 VC 2005 更是默认打开使用 Unicode 模式，而 VC6.0 则使用 ANSI 字符模式。为了使用户开发的程序更为标准、国际化，建议采用 Unicode 模式，同时用户在 Unicode 平台或操作系统中使用 ANSI 字符，系统有一个 ANSI 到 Unicode 的转换动作，额外地消耗 CPU 资源。现在提倡构建节约型社会，开发人员开发的程序也应当顺应历史潮流，少做或不做无用功！

图 7-10 BaseClasses 项目批编译生成

为安全起见，需要首先“清理”旧编译生成，然后再“生成”。根据 VC 2005 的“输出”窗口的提示，查看各个版本的生成是否有错误或警告。如我们发现 Debug Unicode 模式有几

个警告 warning C4996: "_vsnwprintf "被声明为否决的。

STEP 01 切换当前的工作模式为 Debug Unicode，如图 7-11 所示。

STEP 02 手工配置：项目→属性→配置属性→C/C++→预处理器→预处理器定义，在命令行中添加_CRT_SECURE_NO_DEPRECATE。F7 编译后，还有警告，则继续手工配置添加_CRT_NON_CONFORMING_SWPRINTFS。最后出现：

========== 生成: 0 已成功, 0 已失败, 1 最新, 0 已跳过==========

说明程序编译已经大功告成了。

至此 4 个版本的 BaseClasses 静态库 lib 均成功生成，后面用户就可以用它们来开发自己的流媒体应用程序了。另外，一般有开发经验的用户都知道，要让 VC 集成开发环境访问到特定的库或头文件，需要修改 VC 的目录或路径。把要访问的 include 路径和 lib 路径告诉 VC，用户在后面就可以在应用程序中放心使用这些库和头文件了。

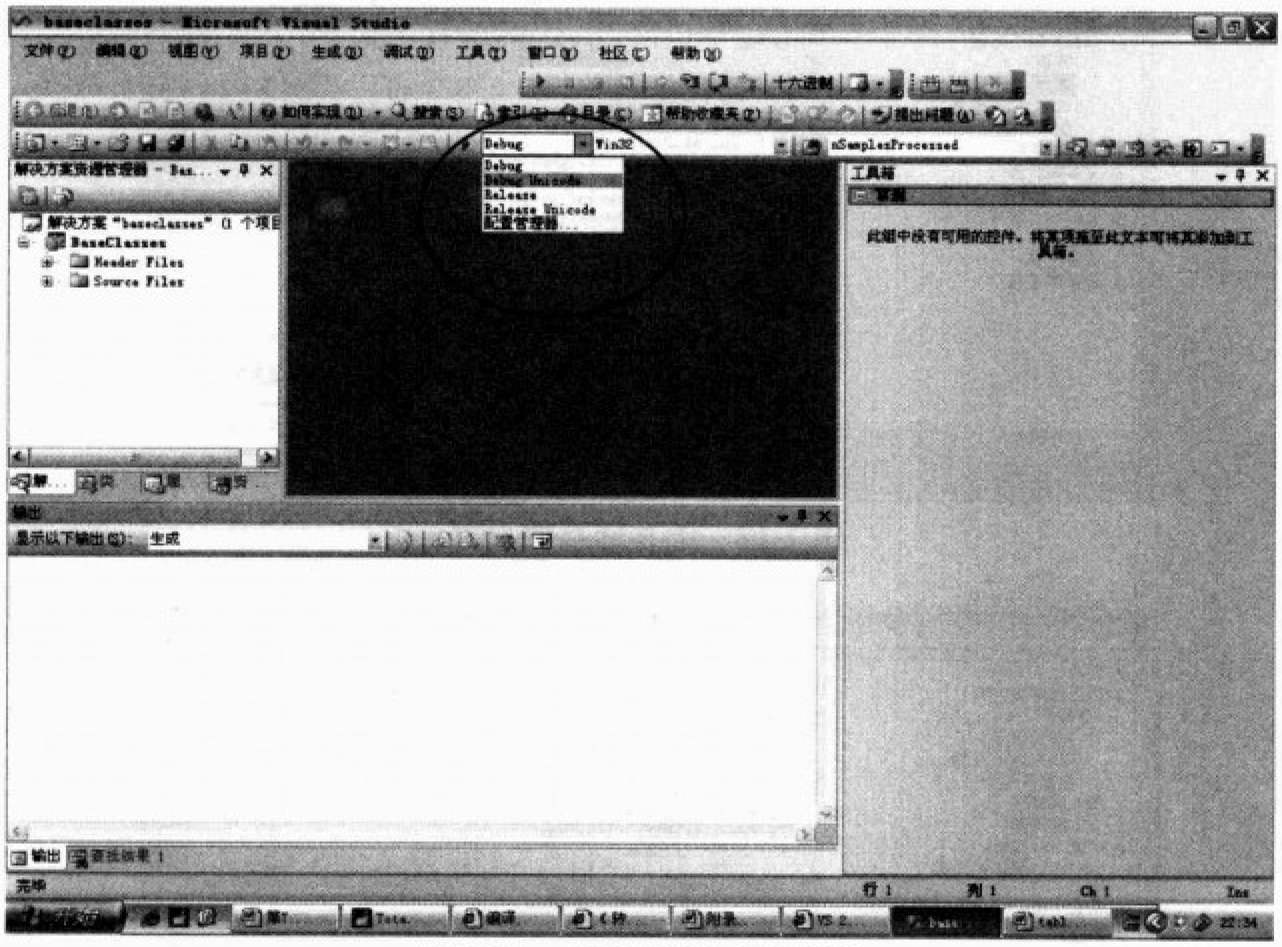


图 7-11　切换当前工作模式

7.2.2　Visual C++开发环境配置

有了 DirectShow SDK 库，用户就可以使用这些库来开发自己的程序了。为了能让 VC++自动搜寻到 SDK 库和头文件，还需要对 VC++的开发环境进行配置。添加库或路径的时候，根据你的要求，添加 Debug、Release、Debug_Unicode、Release_Unicode 版本的库所在路径。下面假定添加非 Unicode 版本的库或路径。

首先确认 VC 2005 是否已经包含了库和头文件所在的路径，因为在安装 VC 2005 时，它会自动添加该目录。如果没有，则需要用户手工添加。

1. 更改添加 include 路径

要添加的 include 内容：

```
D:\DXSDK\Include
D:\DXSDK\Samples\C++\DirectShow\BaseClasses
D:\DXSDK\Samples\C++\Common\Include
```

添加过程如下。

STEP 01 选择“工具”→“选项”命令，出现如图 7-12 所示的“选项”对话框。

STEP 02 选择“包含文件”选项，如果包含的文件没有上面列出的 3 个内容，则需要单击按钮，新插入一行，根据提示添加包含的路径，如图 7-13 所示。

图 7-12　工具选项的设置

图 7-13　选择包含的文件

注意：

在“选项”对话框中，每改变显示的内容的目录，对话框下方提出相应的提示或说明。

2. 更改添加 lib 路径

要添加的 lib 内容：

```
D:\DXSDK\Lib
D:\DXSDK\Samples\C++\DirectShow\BaseClasses\Debug
D:\DXSDK\Samples\C++\DirectShow\BaseClasses\Debug_Unicode
D:\DXSDK\Samples\C++\DirectShow\BaseClasses\Release
D:\DXSDK\Samples\C++\DirectShow\BaseClasses\Release_Unicode
```

添加过程如下。

STEP 01 选择“工具”→“选项”命令，出现如图 7-14 所示的“选项”对话框。

STEP 02 选择“库文件”选项，如果列表中没有上述的库文件路径，则单击 按钮，依次添加用户需要添加的 lib 路径，设置好的库文件属性对话框如图 7-14 所示。

图 7-14 “选项”对话框

3. 添加链接库支持

上面的设置是在 VC 2005 的开发环境的目录（Directories）中，添加用户在开发中可能用到的库或头文件“路径”，需要明确的是文件夹，而不是具体的文件。所以，要使用相关的库支持，还要用户明确地把要使用的库包含、添加到开发环境中。

基于 DirectShow SDK 开发流媒体应用程序，一般需要链接 strmiids.lib 和 quartz.lib，前者定义了 DirectShow 标准的类标识符 CLSID 和接口标识 IID，后者定义了导出函数 AMGetErrorText（如果应用程序中没有使用这个函数，也可以不链接这个库）。

在编译生成 DirectShow 的 BaseClasses 库 strmbasd.lib、STRMBASE.lib 时，由于该工程是生成库而不是应用程序，所以在编译该项目时 VC++ 2005 没有“链接器”选项。但是在开发其他应用可执行程序时，需要添加 DirectShow SDK 库的支持。添加路径：项目→属性→配置属性→链接器→输入→附加依赖项，输入 strmiids.lib quartz.lib，库名之间用空格分开，如图 7-15 所示。另外，在程序中使用 DirectShow SDK 类或接口的代码程序中，还要添加 #include <dshow.h>。

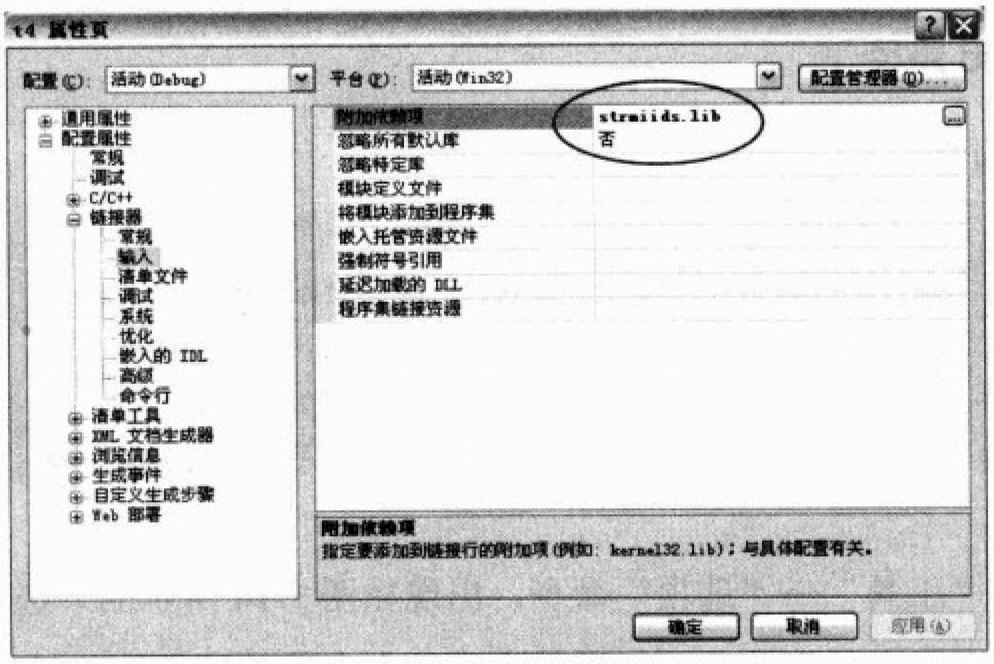


图 7-15 链接器的配置

在添加链接库时，除了以上配置 VC 的开发环境外，也可以在源程序文件开头部分，直接语句编程引入#pragma comment(lib,"strmiids.lib")。

如果程序中没有使用 dshow.h，而是包含了 streams.h，则库文件需要链接 strmbasd.lib、winmm.lib，在源程序文件开头添加：

```
#pragma comment(lib,"strmbasd.lib")
#pragma comment(lib,"winmm.lib")
#include <streams.h>
```

不过，编译器会报出以下的错误。

error C2146：语法错误为缺少“;”（在标识符“m_pString”的前面）。

问题定位在 wxdebug.h(329)中。经分析得知，由于某种原因，编译器认为 PTCHAR 没有定义，那用户可以在类外定义：typedef WCHAR *PTCHAR; 再编译项目。

7.2.3 常见问题解析集锦

在配置 VC++环境进行用户程序开发时，会遇到很多棘手的问题，毕竟 VC 2005 与 VC 6.0 的集成开发环境 IDE 还是有很大的区别的。笔者把配置 VC 2005 开发环境、使用 VC 2005 建立项目过程，DirectShow SDK 的常见问题等列出，并给予分析或解答。

- 为什么我的程序中没有 unicode 模式？当用户在使用项目向导建立项目时，如果为“控制台应用程序”，则只有传统的 Debug 和 Release 模式，没有 Unicode 模式。
- 使用 VC 2005 我能建立哪些项目？首先必须承认 VC 2005 确实比 VC 6.0 强大，向导模板更丰富，如图 7-16 所示的新建项目模板列表。用户可以根据具体的需要定制自己的项目。在 VC 2005 中常用的向导模板一般是“Win32 项目”和“MFC 应用程序”。“Win32 项目”又可细分为 Win 32 应用程序、控制台应用程序、动态库（DLL）和静态库（LIB）。“MFC 应用程序”就是常用的基础类库 MFC 应用程序。当然用户也可以使用“MFC DLL”建立基于 MFC 的动态链接库。其他向导模板使用方法见 MSDN 或其他帮助文档。

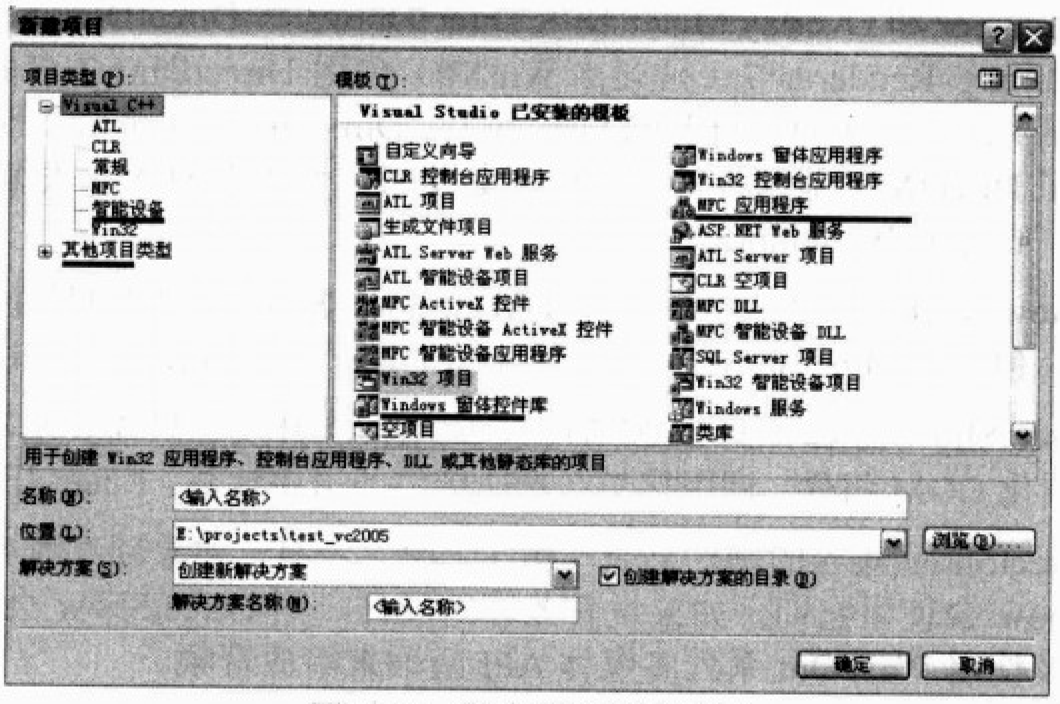


图 7-16　新建项目模板列表

- 为什么我的程序经常出现如下问题？相信使用 VC 2005 的开发人员经常会碰到此问题，自己的程序没有任何错误，且已经编译成功，而在运行时会提示如图 7-17 所示的问题，从而无法运行用户编写的程序。

图 7-17　VC 2005 程序调试问题

由于 Visual Studio 2005 采用了一种新方案，一个 EXE 还要配有一个对应的 manifest 文件（一般在嵌入文件里，所以看不到，不过也可以不嵌入，这样会生产一个<程序名>.exe.manifest 的文件），这是个新功能。微软使用新工具（mt.exe），但是 Fat32 下时间戳有问题（在 NTFS 下就没有这个问题），使 manifest 有时嵌入不到 EXE 中（默认配置是嵌入的，所以就报错找不到 DLL）。

对于上述问题只要把程序目录下两个 Debug 文件下的所有文件删除，然后全部编译生成。笔者建议大家采用这种办法。方法简单有效，配置 VC 属性毕竟麻烦了点！

注意：

一定要先配置这个选项，然后再编译工程。

- DirectShow 支持哪些操作系统？DirectShow 支持 Windows 9X、Windows 2000、Windows Me 和 Windows XP，目前已经支持 Vista。
- 可以使用哪些语言来编写 DirectShow 应用？DirectShow 主要为 C/C++开发设计。Visual Basic 只能使用其中很小的一部分。可以通过 MS JScript 或 VB Script 来支持基于脚本的 DVD 和 TV 应用，也可以用 Delphi 来编写，但 SDK 文档不提供这方面的内容。
- DirectShow 和 DirectX 的其他组件的关系是什么？DirectShow 和 DirectX 的其他组件在内部进行联系。DirectShow 在硬件的支持下使用 DirectSound 和 DirectDraw。

Video Renderer 和 Overlay Mixer 使用 DirectDraw3 和 DirectDraw5 表面（surfaces）。Video Mixing Renderer 7（只支持 WinXP）使用 DirectDraw7 表面。Video Mixing Renderer 9 使用最新的（目前的 Directx 9）Direct3D API 函数。即便是某个应用程序包含了 DirectX 其他组件，用户也不必使用其他组件的 API 去编写它。参考 SDK 的例子：Texture3D Sample。

- DirectShow 只限于多媒体应用吗？DirectShow 默认包含的组件主要是为音/视频流设计的，但是，DirectShow 框架已经成功地用于其他数据流的解决方案中了。
- 使用 DirectShow 需要多少 COM 知识？应用程序开发者只需要基本的 COM 组件知识：实例化 COM 组件、调用接口、管理接口的引用计数。Filter 开发者则需要更多。
- 有与 DirectShow 兼容的硬件列表（HCL）吗？没有。如果硬件兼容 DirectShow，则 DirectShow 会使用它们；如果没有兼容的硬件，则 DirectShow 会使用 GDI 绘制视频，以及使用 WaveOut 系列多媒体 API 函数来播放音频。
- DirectShow 会通过托管代码实现吗？目前还没有这个计划。DirectX SDK 提供了有限的使用音/视频回放类的托管回放功能，用户可以使用 COM interop 创建托管代码的 DirectShow 客户端应用，但是因为性能上的原因，不推荐创建运行在 CLR 上的 filter。
- DirectShow 开发需要什么样的编译器？任何能够产生 COM 对象的编译器都可以。
- DirectShow 与 ActiveMovie 的关系？ActiveMovie 是 DirectShow 原来的名称，现已不再使用，但是一部分 API 仍保留了“AM”的前缀，比如 AM_MEDIA_TYPE 和 IAMVideoAccelerator。
- GraphEdit 工具有源码吗？GraphEdit.exe 是否可再发布？没有源码，不可再发布。
- DMO 可以代替 DirectShow filter 吗？在编写编码器、解码器、效果器应用时，鼓励用 DMO 代替 DirectShow filter。在其他的应用中，使用 DirectShow filter 可能会比较合适。

7.3 DirectShow SDK基本架构

DirectShow SDK 是一个软件开发包，用它来开发音/视频流媒体应用程序，一方面要设计应用程序的界面和事件及响应，另一方面我们要清楚 DirectShow SDK 的工作原理、函数接口内容和一般开发流程。这一节我们详细地阐述 DirectShow SDK 的总体架构、模块组成及开发流程。应用程序的界面设计和事件响应在后续章节中讲解。

7.3.1 DirectShow SDK总体架构

DirectShow SDK 为流媒体的开发带来了极大的便利，它为流媒体应用提供了从底层的视频采集到顶层的流媒体交互应用的解决方案。它是用滤波器的概念实现流媒体的开发，滤波器图表管理器维护滤波器的工作状态。

1. 流媒体革命

在进行流媒体开发时，普遍遇到的问题如下。

- 流媒体中包含海量的、需快速处理的数据流。
- 音/视频流自始至终必须同步，且在同一个速率下播放。
- 流媒体数据有多种源头：本地文件、计算机网络、电视广播和摄像机等。
- 流媒体有多种格式：AVI、ASF、MPEG、DV 等。
- 流媒体程序开发人员事先不清楚终端用户的设备信息等。

2. DirectShow 解决方案

DirectShow 就可以用来解决上述问题。它的设计初衷就是简化在 Windows 平台上开发流媒体应用程序的任务，把应用程序从数据传输、硬件差别和同步机制等复杂问题中分离开来。DirectShow 使用了 DirectSound 和 DirectDraw 技术提高音/视频流媒体的吞吐量，高效的渲染，表现流媒体数据到用户的声卡和显卡。同时，DirectShow 在流媒体数据中封装了时间标签以保证其回放同步。为处理各种可能的流媒体来源、格式和硬件设备，DirectShow 采用了模块化结构，在这些模块中混合、匹配这些应用，而这些模块就是后面经常提到的“滤波器”（Filters）。

DirectShow 使用滤波器支持 WDM 驱动模型采集卡、早期的 VFW 模式采集卡、各种音/视频 CODEC（Audio Compression Manager，ACM；Video Compression Manager，VCM）。下面就是这些应用、DirectShow 组件和 DirectShow 可支持的硬件组件的关系框图，系统框图如图 7-18 所示。

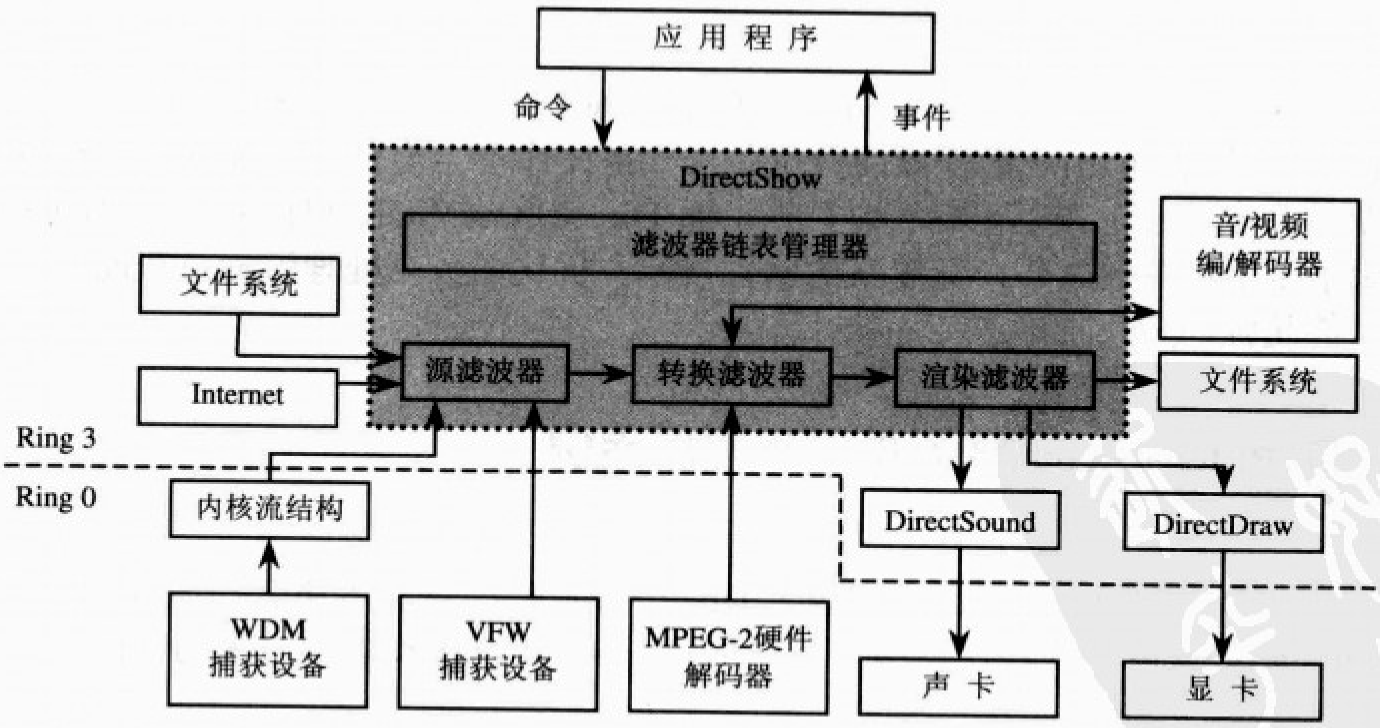


图 7-18 DirectShow SDK 系统框图

框图中的虚线框就是 DirectShow 部分，分析可以看出它起到了承上启下的作用：应用程序根据一定的机制（命令和事件）通过访问 DirectShow 来完成对底层设备视频捕获设备、声卡/显卡、音/视频编/解码器 CODEC 等的访问和维护，而 DirectShow 内部使用滤波器图表管理器管理主要的 3 种滤波器：源滤波器、转换滤波器和渲染滤波器。

3. DirectShow 应用原理

DirectShow SDK 提供了一种开放的开发环境，开发人员可以根据自己的需要定制各种组件。根据 DirectShow 的系统框图不难发现，它使用一种叫滤波器链表（Filter Graph）的模型来管理整个数据流的处理流程，每个部分都是 COM 组件，形成模块化。参与数据处理的模块就是滤波器 Filter。各个滤波器在滤波器链表中按照一定的顺序连成一条流水线，模块间协同工作。DirectShow 给开发人员提供了一系列的常用的标准滤波器用于应用开发，另外研发、开发人员可根据实际情况，定制、开发自己的滤波器来扩展 DirectShow 的应用。据系统框图，DirectShow 的滤波器根据功能主要分为 3 类：源滤波器、转换滤波器和渲染滤波器。

- 源滤波器主要负责从文件、网络或各种 WDM/VFW 驱动模式的视频捕获设备、数字照像机 DC、数字摄像机 DV 等获取数据。
- 转换滤波器主要负责数据格式的转换，例如我们经常说的音/视频编/解码、数据流（音频和视频）的分离或合成、特定的处理如图像增强、滤波等。
- 渲染滤波器主要负责数据的最终流向，送到显卡显示或声卡播放，把处理结束的最后结果存储到文件中去。

滤波器一般由一个或多个引脚 Pin 组成，滤波器之间通过引脚相互连接，从而构成一条顺序的链路。根据滤波器的功能，源滤波器仅含有输出引脚；转换滤波器既有输出引脚又有输入引脚；而渲染滤波器仅有输入引脚。DirectShow 的数据处理实际上就是把这些不同功能的滤波器连接起来，依次对数据进行相应的处理，然后再传递给下一级滤波器。

基于 DirectShow SDK 开发的应用程序通过“命令”（command）和“事件”（event）来与 DirectShow 通信。用户调用 DirectShow 的函数对 DirectShow 发命令如打开文件、枚举系统设备、采集音/视频、回放数据等，DirectShow 根据设备和系统的状态以事件的方式通知应用程序。

DirectShow SDK 为我们开发相关的音/视频应用程序极大地降低复杂度，如驱动中的 Ring 概念。如果直接访问音/视频硬件设备，一是操作不当会使系统崩溃甚至硬件损坏，二是前后要做许多辅助工作以正确获取数据。而 DirectShow SDK 则在应用层 Ring3 和驱动层 Ring0 做了承上启下的工作，其丰富的函数接口、优化的数据处理流程可帮助开发人员迅速开发出功能超强的应用程序。

7.3.2 DirectShow SDK简单应用程序案例

通过上一节的学习，读者已经掌握了 DirectShow 的一些基本概念和原理，现在开始编写第一个 DirectShow 应用程序，体验一下使用 DirectShow SDK 编程带来的快捷、方便和强大功能。

一个 DirectShow 应用程序是由一个个称为滤波器 filter 的 COM 构件组合而成的，filter 执行一些多媒体流的操作，如读文件、从视频采集设备中获得视频、将不同的格式的流解码、将数据送到显卡或声卡中去。

转换 filter 接收输入并产生输出。举个例子，一个解码 MPEG2 视频流的 filter，输入 MPEG2 格式的视频流，输出是一系列未压缩的视频图像帧。

在 DirectShow 中，应用程序要实现这些功能就必须将这些 filter 链接在一起，因此一个 filter 的输出就变成了另一个 filter 的输入，这一系列串在一起的 filter 称为滤波器链表（filter graph）。例如，图 7-19 就显示了一个播放 AVI 文件的 filter graph。

File Source(Async) filter 从硬盘中读取 AVI 文件；AVI Splitter filter 分析文件将其分离成两个流：一个压缩的视频流和一个音频流；AVI Decompressor filter 将视频流解码；Video Renderer filter 将解码后的视频帧通过 DirectDraw 或 GDI 显示出来；Default DirectSound Device filter 使用 DirectSound 播放音频流。

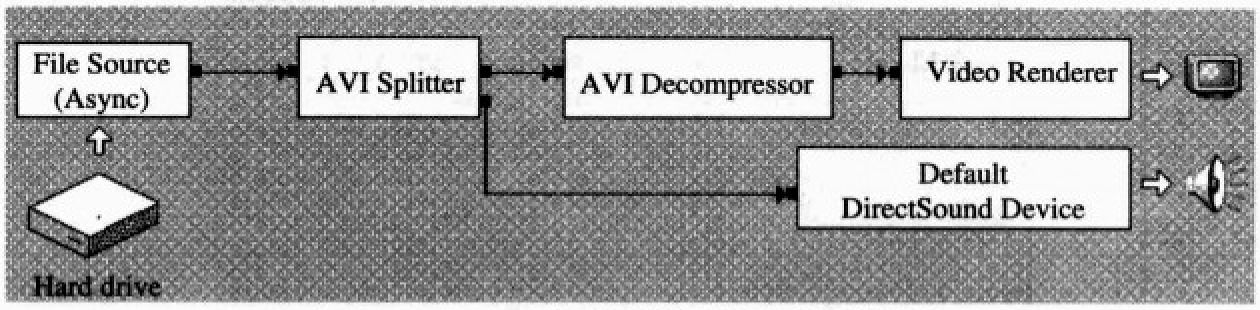


图 7-19　AVI 文件播放滤波器链表框图

应用程序没有必要对这些数据流进行直接管理，而是通过一个叫滤波器链表管理器（Filter Graph Manager）的上层组件来控制这些 filters。应用程序调用上层 API 如“Run”（播放）或“Stop”（停止）。如果用户需要对数据流做更多的操作，则可以通过 COM 接口直接进入 filter。Filter Graph Manager 同样也输出事件通知给应用程序。Filter Graph 的另一个用途是将 filter 连在一起创建一个 filter graph。

编写一个 DirectShow 应用程序大体需要 3 个步骤，图 7-20 展示了这 3 个步骤的任务。

STEP 01 创建一个 Filter Graph Manager 的实例。

STEP 02 使用 Filter Graph Manager 创建一个 filter graph，此时需要必备的 filter。

STEP 03 使用 Filter Graph Manager 控制 filter graph 和通过这些 filter 的流，在这个过程中，应用程序会收到 Filter Graph Manager 发送的事件。

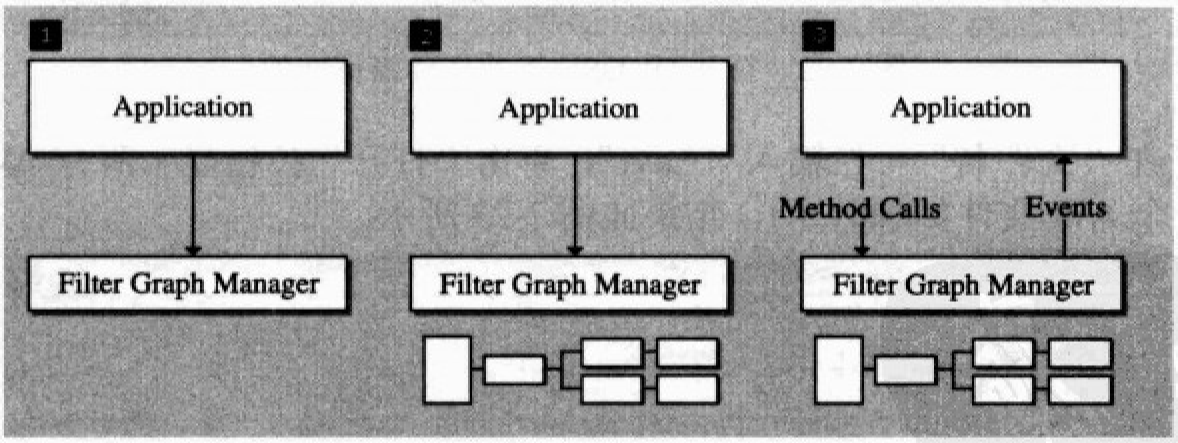


图 7-20　编写 DirectShow 应用程序

完成以上的几个步骤后，应用程序最后需要释放这个 Filter Graph Manager 和所有的 Filter 资源。

1．使用 GraphEdit.exe 程序播放 AVI 文件

前面介绍了 DirectShow SDK 的流媒体应用的特点及其组成部分。为能够快速感受 DirectShow 的流媒体应用的强大功能、验证流媒体开发框架的可行性，微软公司提供了一个

简单的应用程序。在 DirectX SDK 安装目录 D:\DXSDK\Bin\DXUtils\下，有一个名为 graphedt.exe 的应用程序。它是微软提供的“滤波器链表编辑软件”，一方面可以直接打开媒体文件，程序自动剖析构造滤波器链表；另一方面可以逐个添加各种滤波器，然后手工连接，形成一个滤波器链表。在这里使用它的第一种功能，直接打开媒体文件。

STEP 01 运行 graphedt.exe，如图 7-21 所示。

STEP 02 打开文件 D:\DXSDK\Samples\Media\，打开路径如图 7-22 所示。

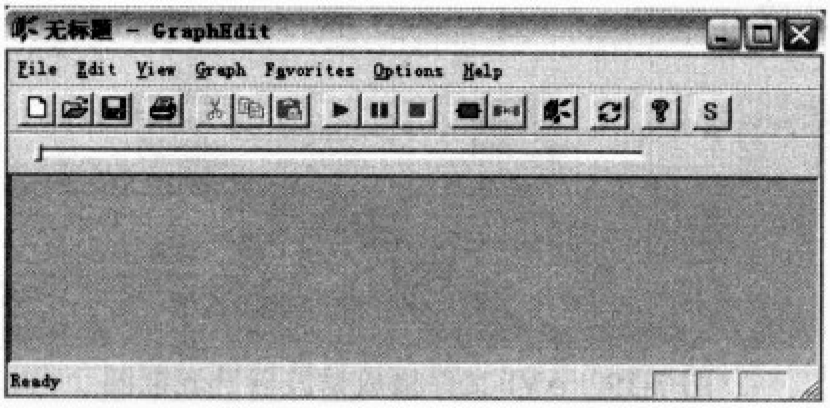


图 7-21　GraphEdit 应用程序

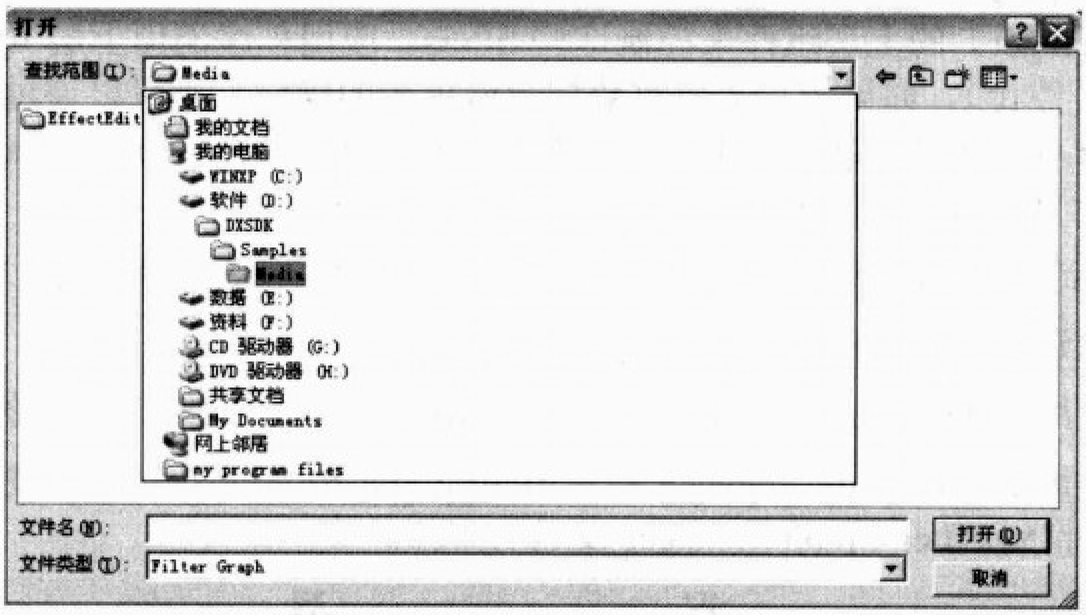


图 7-22　打开 DXDSK 下的媒体媒体文件

STEP 03 在“文件名”一栏输入“*.avi”，单击“打开”按钮显示所有 AVI 文件名。然后选择其中一个 AVI 文件如 ruby.avi，结果如图 7-23 所示。

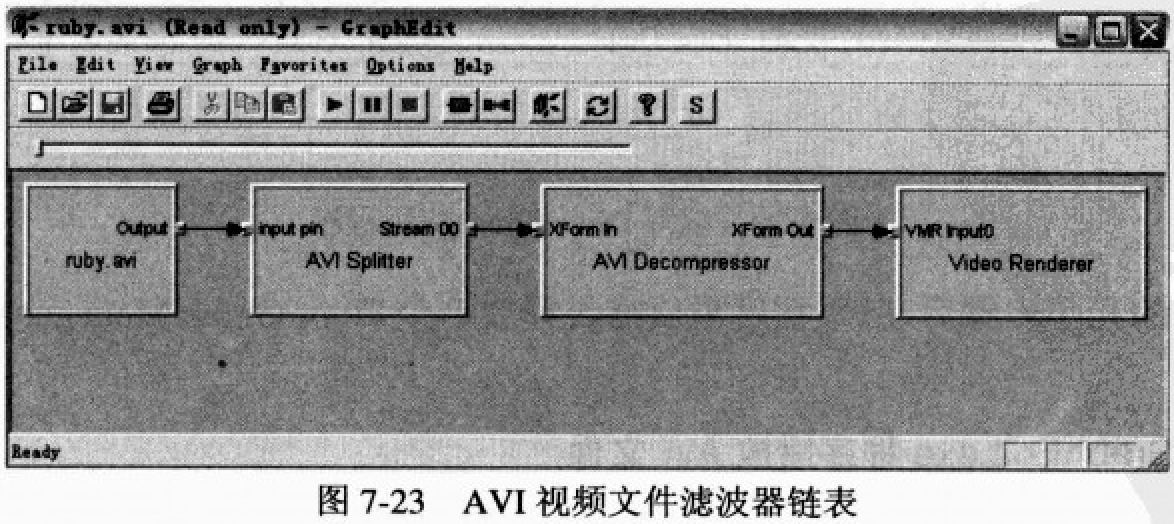


图 7-23　AVI 视频文件滤波器链表

根据已有的知识，由图 7-24 可以判断出 ruby.avi 文件只有视频，没有音频流。从左至右

依次为前面介绍过的源滤波器（File Source）、转换滤波器又包括 AVI 分离器（AVI Splitter）、AVI 解码器（Decompressor）和渲染滤波器（Video Renderer），它们分别负责数据的读取、转换或分离、显示等任务。

STEP 04 播放 AVI 文件，选择“Graph”→“Play”命令，开始运行播放，如图 7-24 所示。

STEP 05 回放后的图像如图 7-25 所示，当停止后，用户还可以单击“▶|”按钮，继续回放。

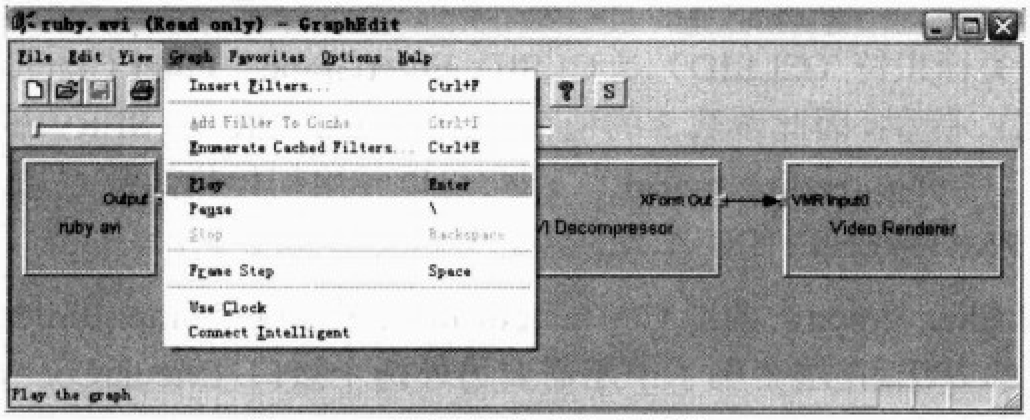


图 7-24　滤波器链表控制播放 Play

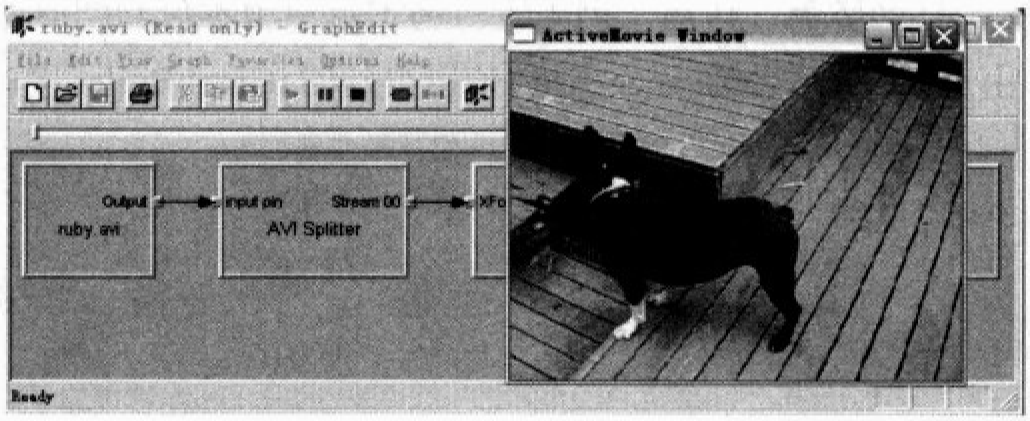


图 7-25　正在播放 AVI 文件

至此读者会发现使用 GraphEdit.exe 程序可以很清楚地掌握媒体文件的格式，以及 DirectShow 使用了哪些滤波器，是如何连接的等。对于每个滤波器的前后顺序及功能，读者有了一个感性认识。到后面的案例中，还会使用它来展示它的方便和快捷。

下面就介绍如何编程实现上述任务。

2．编程实现 AVI 文件的播放

现在介绍使用 VC++ 2005 进行编程，实现使用 DirectShow SDK 来播放 AVI 文件的任务。

1）VC++ 2005 开发环境配置

前面已经详细地说明了使用 VC 2005 和 DirectShow 开发流媒体程序时的各种开发环境配置。其实主要的是链接库的添加，具体见 7.2.2 一节。

2）DirectShow 开发步骤

现在就用最简单的“控制台应用程序”模式，用最少的代码实现回放，展示 DirectShow 编程的强大功能。

正如上一节所讲的创建 DirectShow 应用程序的 3 个步骤，第一步，调用 CoInitialize 来初始化，然后调用 CoCreateInstance 创建 Filter Graph Manager。

```
  HRESULT hr = CoInitialize(NULL);         /*初始化COM库*/
    if (FAILED(hr)) {                      /*若初始化失败则返回*/
        return;
    }
   IGraphBuilder *pGraph;                  /*滤波器图表管理器*/
   HRESULT hr = CoCreateInstance(CLSID_FilterGraph, NULL,
         CLSCTX_INPROC_SERVER, IID_IGraphBuilder, (void **)&pGraph); /*创建滤波
器图表管理器*/
```

上述代码中，类标识符（CLSID）是 CLSID_FilterGraph。Filter Graph Manager 由进程内 DLL(in-process DLL)提供，因此参数 3 的值为 CLSCTX_INPROC_SERVER。由于 DirectShow 运行自由线程模式（free-threading model），用户可以使用初始化函数 CoInitializeEx，其中参数使用 COINIT_APARTMENTTHREADED。

第二步是创建 filter graph，调用 CoCreateInstance 得到的 IGraphBuilder 接口包含了大部分创建 filter graph 的方法。在这个例子中还需要另外两个接口：IMediaControl 和 IMediaEvent。

IMediaControl 控制数据流，它包含开启和停止 graph 的函数；IMediaEvent 包含从 Filter Graph Manager 获取事件的函数，在这个例子中，这个接口用来得到回放结束事件。

所有这些接口由 Filter Graph Manager 提供，使用得到的 IGraphBuiler 接口指针来查询得到。

```
   IMediaControl *pControl;               /*媒体控制接口*/
   IMediaEvent   *pEvent;                 /*媒体事件接口*/
   hr=pGraph->QueryInterface(IID_IMediaControl, (void **)&pControl);
                                                  /*在管理器下获取媒体控制接口*/
   hr=pGraph->QueryInterface(IID_IMediaEvent, (void **)&pEvent;
                                                  /*在管理器下获取媒体事件接口*/
```

现在用户可以创建 filter graph 了，对于文件回放只需要一个简单的调用：

```
  hr = pGraph->RenderFile(L"D:\\DXSDK\\Samples\\Media\\ruby.avi", NULL);
```

IGraphBuilder::RenderFile 方法创建了一个能够播放指定文件的 filter graph，事实上，原本需要做的一些如创建 filter 实例及将这些 filter 连接起来的工作，都由这个函数自动完成了，如果是视频文件，则这个 filter graph 看起来应该是这个样子：[file source]->[如果是压缩格式，则这里是个解码器]->[Video Renderer]。

如要开始回放，则调用 IMediaControl::Run 方法：

```
 hr = pControl->Run();
```

当 filter graph 运行时，数据经过各个 filter 最后回放为视频或音频。回放发生在一个单独的线程内。用户可以通过调用 IMediaEvent::WaitForCompletion 方法来等待回放的结束。

```
   long evCode = 0;                               /*事件码*/
   pEvent->WaitForCompletion(INFINITE, &evCode); /*无限时间的等待，直到媒体播放结束*/
```

这个方法在播放期间被阻塞，直至播放结束或超时。

当应用程序结束时，需要释放接口指针并关闭 COM 库。

```
pControl->Release();             /*释放媒体控制接口*/
pEvent->Release();               /*释放媒体事件接口*/
```

```
pGraph->Release();                /*释放图表管理器接口*/
CoUninitialize();                 /*反初始化COM库*/
```

3）编程实现及源代码

通过上一节我们掌握了使用 DirectShow 做流媒体文件回放时的过程和原理。现在我们通过建立 Visual C++ 2005 项目、编辑源代码，来完成整个工作。

STEP 01 启动 Visual C++ 2005。选择“开始”→“运行”命令，在弹出的对话框中输入“devenv”。

STEP 02 启动项目向导：选择“文件”→“新建”命令，在打开的对话框中单击“项目”按钮，然后选择 Win32 选项，如图 7-26 所示。

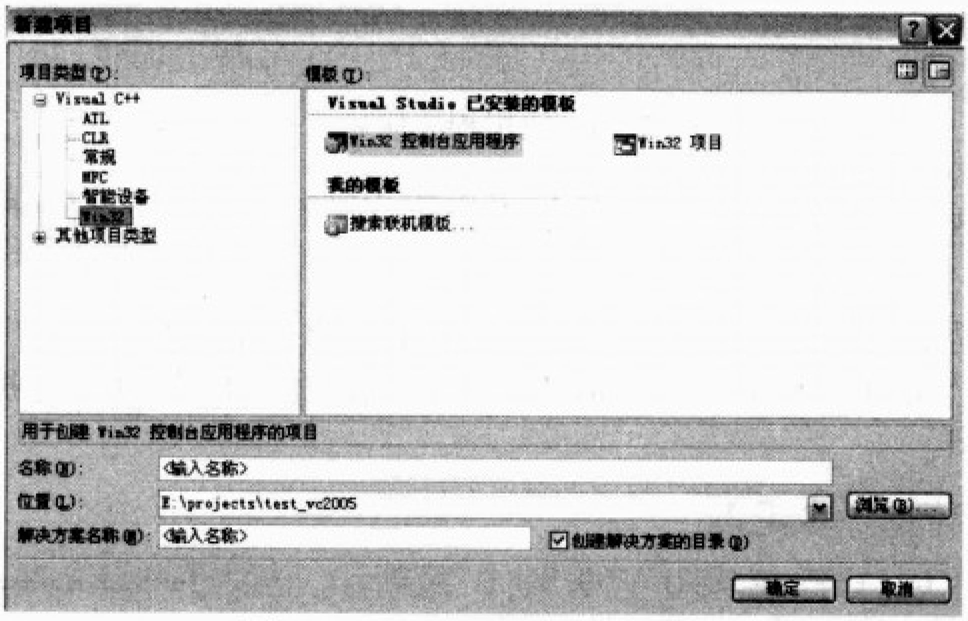


图 7-26　Win32 项目建立向导

现在的任务是使用简单的传统的“Win32 控制台应用程序”模式建立程序。在名称栏中输入 aviplayer，目录用户可以自己更改，然后单击“确定”按钮，出现如图 7-27 所示的画面。

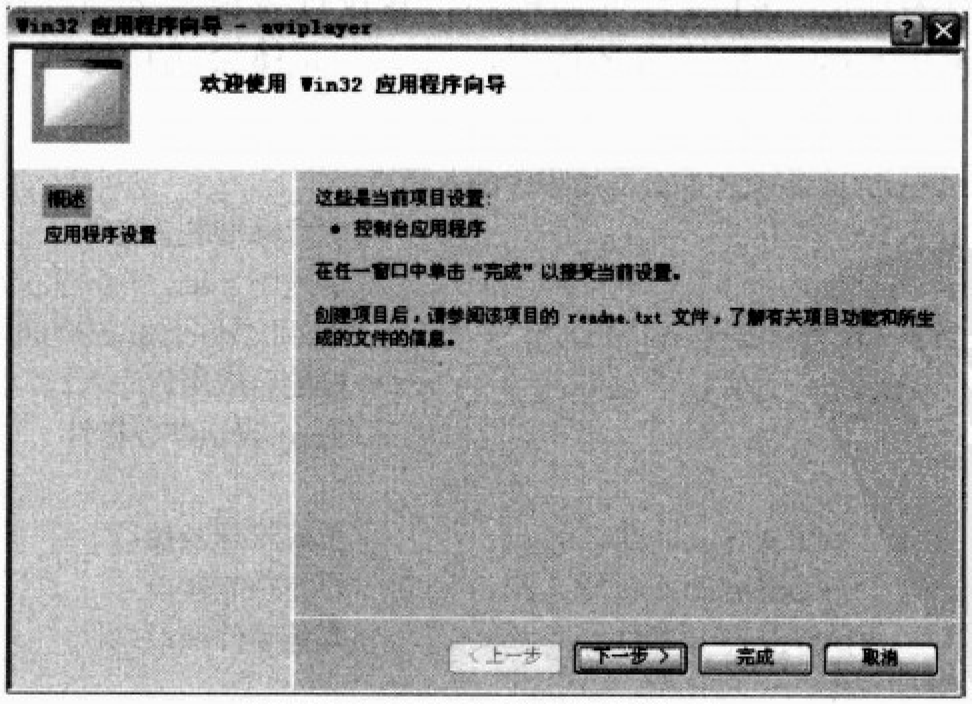


图 7-27　Win32 应用程序向导

STEP 03 在图 7-28 中，选中“控制台应用程序”单选按钮，完成 Win32 应用程序向导。

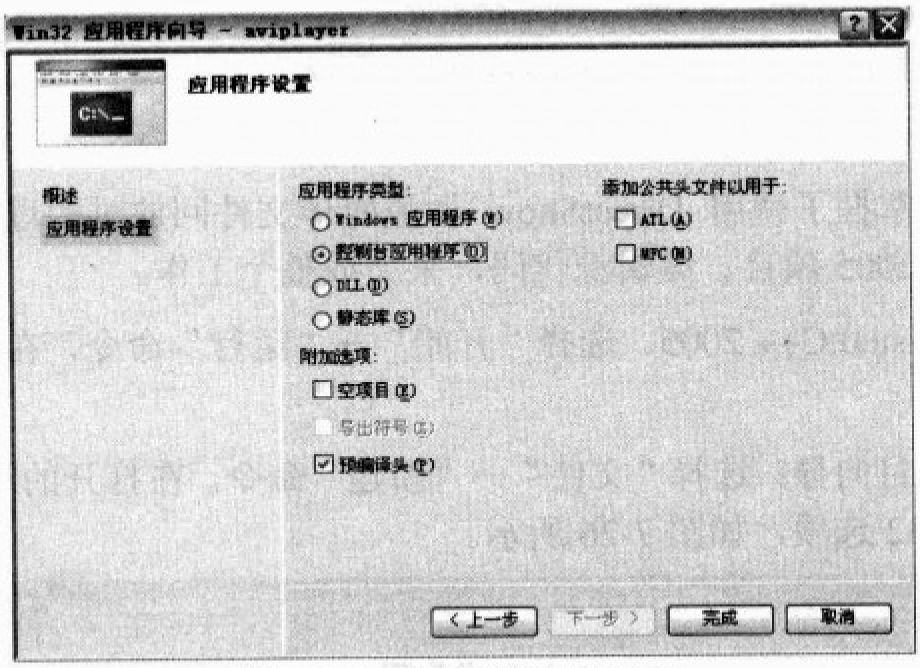


图 7-28　应用程序向导选择

单击“完成”按钮，至此 VC 2005 就帮用户建立了控制台应用程序所必须的源文件和预编译头文件。

STEP 04 编译当前项目，确保项目设置、编译器能正确工作。用户可以在“return 0;”语句前添加“printf("this is a test for vc2005!\n");”。然后按“F7”键开始编译生成解决方案。确认出现以下的提示：

aviplayer - 0 个错误，个警告

========== 生成: 1 已成功, 0 已失败, 0 最新, 0 已跳过==========

在行“return 0;”前设置断点，按“F9”键。然后运行，按“F5”键。如果程序能运行到断点处，且已经弹出控制台窗口，内容为 this is a test for vc2005。这样用户的 VC 2005 平台一切正常！可以按“shift + F5”组合键停止运行。

下面是编程实现的播放 AVI 文件的源程序，建议读者根据下面的内容自己输入到电脑中，一方面体会 DirectShow 函数的命名特点，另一方面体会成功运行出自己的编写程序的那份快乐。

```
  //aviplayer.cpp : 定义控制台应用程序的入口点。
#include "stdafx.h"                          //包含标准的头文件
#include <dshow.h>                           //引入DirectShow头文件
#pragma comment(lib,"strmiids.lib")          //引入DirectShow的库文件
int _tmain(int argc, _TCHAR* argv[])         //控制台应用程序入口，argc表示参数个数，
                                             //argv表示参数指针

{
    IGraphBuilder *pGraph = NULL;            //图表管理器接口
    IMediaControl *pControl = NULL;          //媒体控制接口
    IMediaEvent   *pEvent = NULL;            //媒体事件接口
    //初始化COM库
    HRESULT hr = CoInitialize(NULL);
    if (FAILED(hr))
    {
        printf("ERROR - Could not initialize COM library");
```

```
        return -1;
        }
    //创建滤波器链表管理器
    hr=CoCreateInstance(CLSID_FilterGraph, NULL,
            CLSCTX_INPROC_SERVER,  IID_IGraphBuilder, (void **)&pGraph);
    if (FAILED(hr))
    {
        printf("ERROR - Could not create the Filter Graph Manager.");
            return -1;
     }
      //查询媒体控制和媒体事件接口
    hr = pGraph->QueryInterface(IID_IMediaControl, (void **)&pControl);
    hr = pGraph->QueryInterface(IID_IMediaEvent, (void **)&pEvent);
    //建立链表，在这里用户可以更改待播放的文件名称
    hr = pGraph->RenderFile(L"D:\\DXSDK\\Samples\\Media\\ruby.avi", NULL);
    if (SUCCEEDED(hr))
        {
            //运行链表
            hr = pControl->Run();
            if (SUCCEEDED(hr))
            {
                //等待回放结束事件
                long evCode;
                pEvent->WaitForCompletion(INFINITE, &evCode);
        //切记：在实际应用当中，不能使用INFINITE标识， 因为它会不确定的阻塞程序
            }
        }
    //释放所有资源和关闭COM库
        pControl->Release();
        pEvent->Release();
        pGraph->Release();
        CoUninitialize();
    printf("this is a test for vc2005!\n");
    return 0;
    }
```

前面已经介绍过，使用 DirectShow SDK 开发流媒体应用程序，除了要包含头文件 dhsow.h 外，还需把必要的库链接到程序中。方式有两种，配置开发环境和源程序引入。这里使用源程序直接引入的方法，在包含头文件语句之后、入口_tmain 函数之前增加以下语句：

```
#pragma comment(lib,"strmiids.lib")。
```

注意：

该语句不能放置在源文件的最开始。

至此源文件已经输入，DirectShow SDK 链接库、头文件已经引入，按“F7”快捷键编译生成。如果没有错误，则按“F5”快捷键运行，可能会有一个警告，可以不必理会，运行结

果如图 7-29 所示。

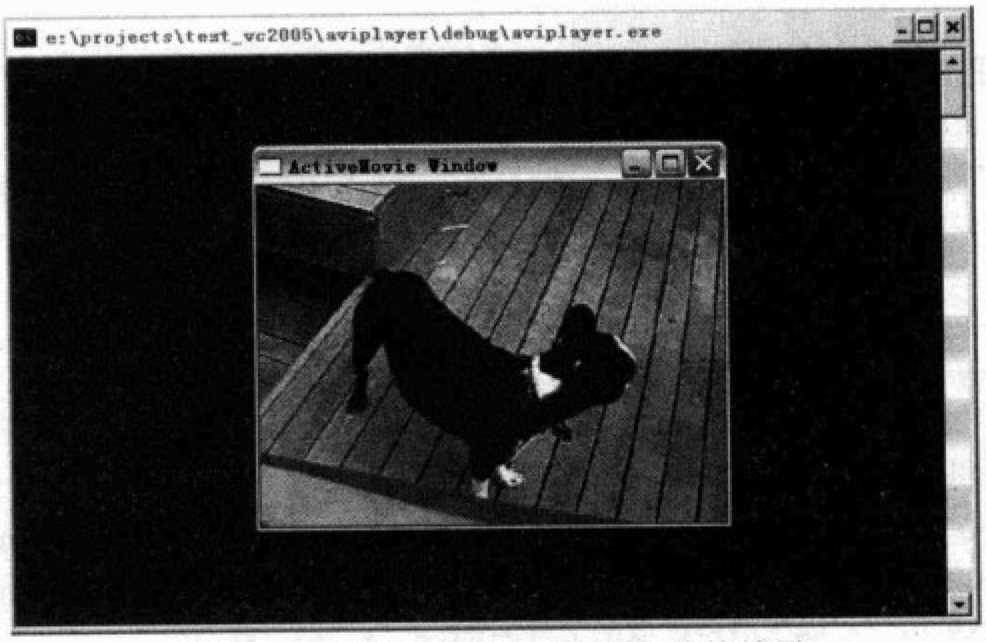


图 7-29　控制台模式回放 AVI 文件结果

通过以上 AVI 文件的播放案例，初步体会了基于 DirectShow SDK 的流媒体应用。使用 GraphEdit.exe 直接播放媒体文件，剖析并展示了所用到的各种 filter 组件，清晰地展示了流媒体的数据格式和用到的解码器等组件。编程实现流媒体应用是开发人员的热衷的目标，而要编写出强大的流媒体程序，需要对各种 filter 组件和管理这些 filter 组件的滤波器链表 Filer Graph 及相关知识详细地了解和掌握。

7.3.3　滤波器链表Filer Graph及各种组件

本节主要描述 DirectShow 的主要组件，为 DirectShow 应用程序和 DirectShow Filter 开发者提供一个介绍。应用程序开发者可以忽略掉很多底层部分，但是，了解底层对于理解 DirectShow 架构还是很有帮助的。

1．滤波器（Filter）

DirectShow 使用一个模块化的架构，每个处理过程都由一个叫做 filter 的 COM 对象来实现。DirectShow 为应用程序提供了一系列标准的 filters 来完成流媒体的开发，当然开发者也可以根据需要编写自己的 filter 来扩展 DirectShow 的功能。下面仍然以播放一个 AVI 文件为例说明 filter 的工作过程。

STEP 01 源滤波器（File Source）以异步方式从文件中读取数据并转成位流，然后传送到后续的转换滤波器。

STEP 02 转换滤波器包括数据分离器（AVI Splitter）和音/视频解码器（AVI Decompressor），他们负责检查 AVI 文件头，分析位流并将其分离成视频流和音频流。

STEP 03 视频解码器将视频解码（不同的压缩格式使用不同的解码 filter）。

STEP 04 视频渲染滤波器（Video Renderer filter）将视频图像送显示设备显示出来。

STEP 05 音频渲染滤波器（Default DirectSound Device filter）将音频送入声卡。

如图 7-30 所示，每个 Filter 与一个或多个其他的 Filter 相连，其中的连接点也是一个 COM

对象，称做引脚 Pin。Filter 使用 Pin 将数据从一个 Filter 转移到另一个 Filter，图中的箭头指示了数据流动的方向。在 DirectShow 中，这一系列连接在一起的 Filter 称做 Filter Graph。

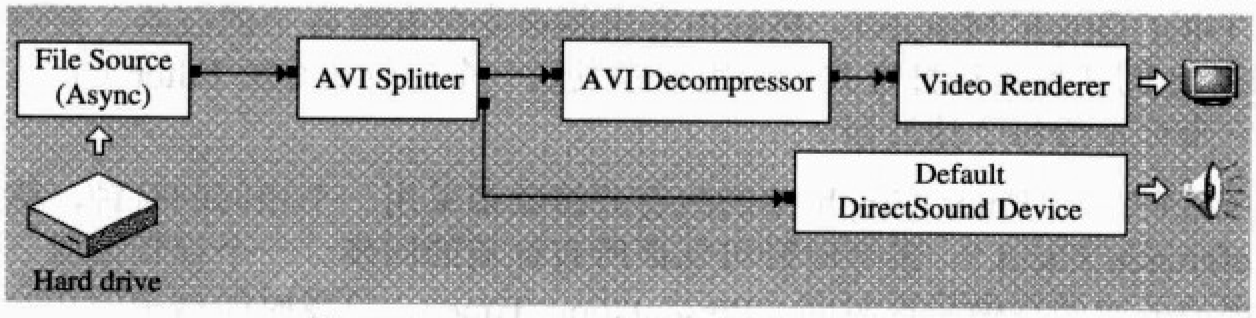


图 7-30 DirectShow SDK 回放 AVI 框架

任何 Filter 都可能处于 3 种不同的状态：运行、停止和暂停。Filter 在“运行”状态时处理数据，在“停止”状态时停止处理数据，“暂停”状态则是表示就绪并可以开始进入运行状态。除了极个别的情况，一个 Filter Graph 链表中的所有 Filter 通常都处于同一个状态下，因此，Filter Graph 也可以称其处于运行、停止和暂停状态。

在通常情况下，Filter 可以根据其在链表中位置或处理的数据状态，分为以下几类。

- **Source Filter ——源滤波器**：Filter Graph 的数据源，这些数据可以来自硬盘文件、网络数据流、摄像头或任何其他音频、视频采集设备。每一个 Source Filte 处理不同类型的数据源。典型的如 File Source Filter、USB 摄像头。
- **Transform Filter——转换滤波器**：用来转换数据，获取输入流，处理数据，生成输出流并将它送入下一个 Filter。编码 Filter 和解码 Filter 都属于这个种类。
- **Renderer Filter——渲染滤波器**：处于 Filter 链的最后一级，接受数据并将其表现、渲染给用户。比如，一个视频渲染器 Renderer 到显卡，从而在显示器上绘制视频图像；一个音频渲染器 Renderer 将音频数据送入声卡；一个写文件 Filter（File Writer）将数据存盘。
- **Splitter Filter——分离滤波器**：分析输入的数据流并将其分解成 2 路或多路，比如 AVI Splitter 分析位流并将其分解成视频流和音频流。
- **Mux Filter——复用滤波器**：将多路输入流合并成一路单独的数据流。比如，AVI Mux 正好与 AVI Splitter 做相反的工作，它是将视频和音频流合成为一个 AVI 格式的位流。

当然以上的分类并不是绝对的，比如，ASF 读滤波器（ASF Reader Filter）同时充当了 Source Filter 和 Splitter Filter 的角色。

2. 滤波器链表管理器（Filter Graph Manager）

Filter Graph Manager 是一个用以控制 Filter Graph 中的 Filter 的 COM 对象。它提供了许多功能，包括以下几项。

- 协调滤波器之间的状态变化，使其一致。
- 建立参考时钟（Reference Clock）。
- 将滤波器的事件、消息返回给应用程序。
- 为应用程序提供建立 Filter Graph 的方法。

这里详细地描述上述功能。

- 状态变化：所有滤波器的状态变化应该是一致的，即必须遵循一个特定的次序，因此，应用程序不能直接将状态变化的命令发给滤波器，而是先将一个简单的命令发给 Filter Graph Manager，然后由它来将命令分发给各个滤波器。查询、定位命令 Seeking 同样使用这种方式，应用程序发送一个定位命令给 Filter Graph Manager，由它来分发给每个滤波器。
- 参考时钟：在 Filter Graph 中的所有滤波器都采用一个相同的时钟，称为参考时钟（Reference Clock）。参考时钟可保证所有位流的同步。一个视频帧或一段音频样本被播放的时间称做呈现时间（Presentation Time），呈现时间精确地相对于参考时钟来确定。Filter Graph Manager 选择的参考时钟可以是声卡上的时钟或系统时钟。
- Graph 事件：Filter Graph Manager 采用事件机制，类似于 Windows 中使用一个消息循环队列来通知应用程序发生在 Filter Graph 中的事件。
- Graph-buliding 方法：Filter Graph Manager 提供给应用程序将各个滤波器加入到 Filter Graph 中的方法，以及将滤波器与滤波器连接或断开连接的方法。Filter Graph Manager 没有提供操纵滤波器之间数据流动的方法，这个功能由滤波器通过引脚来独立完成。

3. 媒体类型（Media Type）

DirectShow 是模块化工作的，则模块之间数据匹配须一致，因此需要有一个方式来描述在 Filter Graph 各个点之间的数据格式。比如，AVI 回放，数据输入是一个 RIFF 块的流，然后被分离成音频和视频流。视频流由一系列的被压缩的视频帧组成，解压后，视频流又变成了一系列未压缩的位图，音频与视频类似。DirectShow 使用媒体类型 Media Type 来检查数字媒体格式，当两个滤波器连接时，它们需要协商决定同一个 Media Type。Media Type 标识了从上一个滤波器递交到下一个滤波器或物理层的数据流格式，如果两个滤波器对 Media Type 不能协商一致，则不能进行连接。

对于某些应用程序，用户可不必去关心 Media Type，比如文件回放，DirectShow 做了所有有关媒体类型匹配的工作。

Media type 使用 AM_MEDIA_TYPE 结构体来定义，这个结构体包含了以下内容。

- **majortype：** 主类型，是一个 GUID，定义了数据的整体类型，包括了视频、音频、未分析的字节流、MIDI 等。
- **subtype：** 子类型，另一个 GUID，进一步定义了数据格式。比如，如果主类型是视频，则子类型可以是 RGB-24、RGB-32、UYVY 等格式，如果主类型是音频，则可能是 PCM 或 MPEG-1 payload 等。子类型提供了比主类型更多的内容，但仍未提供完整的格式定义，比如，子类型没有定义图像尺寸和帧率，这些都将在下面的 format block 中被定义。
- **format block：** 格式块 pbFormat，定义了具体的格式。格式块是 AM_MEDIA_TYPE 结构体中一个单独被分配的内存空间，pbFormat 成员指向这块内存空间。因为不同的格式会有不同的格式描述，所以 pbFormat 成员的类型是 BYTE *。比如，PCM 音频使用 WAVEFORMATEX 结构体，视频使用不同的结构体，包括 VIDEOINFOHEADER 和 VIDEOINFOHEADER2。formattype 成员是一个 GUID，指定了格式块包含了哪种结构体，每一种格式的结构体都被分配了 GUID。cbFormat

成员定义了格式块的长度。

当格式块被定义时，主类型和子类型包含的信息就显得有点多余了。其实，主类型和子类型为识别格式提供了一个便利的方法，比如，用户可以指定一个普通的 24 位 RGB 格式(MEDIASUBTYPE_RGB24)，而不需要去关心 VIDEOINFOHEADER 结构体中诸如图像尺寸和帧率这些信息。

```
typedef struct  _MediaType {
    GUID      majortype;                //主类型标识
    GUID      subtype;                  //子类型标识
    BOOL      bFixedSizeSamples;        //样本大小是否固定
    BOOL      bTemporalCompression;     //是否使用帧间压缩
    ULONG     lSampleSize;              //样本大小，若压缩则为0
    GUID      formattype;               //媒体格式块标识
    IUnknown  *pUnk;                    //未使用
    ULONG     cbFormat;                 //格式块标识大小
    BYTE      *pbFormat;                //指向格式块
} AM_MEDIA_TYPE;
```

- **IsampleSize**：如果这个字段非零，则表示这是每个样本的尺寸，对于压缩的数据则是零，表示样本的尺寸会改变。
- **BFixdSizeSamples**：如果为 TRUE，则表示 ISampleSize 有效，否则失效。
- **BTemporalCompression**：如果是 FALSE，则表示所有帧都是关键帧，否则表示使用了时域编码或帧间编码。

现在通过一个例程来看滤波器如何检测媒体类型。

```
HRESULT CheckMediaType(AM_MEDIA_TYPE *pmt)
{
    if (pmt == NULL) return E_POINTER;
//检查主类型，我们需要的是视频
if (pmt->majortype != MEDIATYPE_Video)
  {
  return VFW_E_INVALIDMEDIATYPE;
    }
//检查子类型，我们需要的是24bit RGB
    if (pmt->subtype != MEDIASUBTYPE_RGB24)
    {
   return VFW_E_INVALIDMEDIATYPE;
    }
//检查format type和格式块的大小
if ((pmt->formattype == FORMAT_VideoInfo) &&
          (pmt->cbFormat >= sizeof(VIDEOINFOHEADER) &&
          (pmt->pbFormat != NULL))
    {
        //现在可以安全地将格式块指针指向正确的结构体
        VIDEOINFOHEADER *pVIH = (VIDEOINFOHEADER *)pmt->pbFormat;
          //检查pVIH (未用)，如果正确，返回S_OK
```

```
        return S_OK;
    }
  return VFW_E_INVALIDMEDIATYPE;
 }
```

另外在调试中，为获取特定媒体格式的详细信息，可使用 DisplayType 方法。该函数的原型如下。

```
void DisplayType(
   LPTSTR label,                    //即将显示媒体信息的字符串
   const AM_MEDIA_TYPE *pmtIn       //指向AM_MEDIA_TYPE结构体
);
```

AM_MEDIA_TYPE 结构包含一个指向数据块的指针，因此，当你使用这个结构的时候，一定要特别注意内存分配，防止内存泄露发生。

下面是与 Media Type 相关的几个函数。

1）内存分配（3 个函数）

```
AM_MEDIA_TYPE * WINAPI CreateMediaType(
AM_MEDIA_TYPE const *pSrc);
```

功　能	分配一个新的 AM_MEDIA_TYPE 结构，包含上面提到的格式块。释放由该函数分配的内存，则调用 DeleteMediaType 函数
参数说明	参数 pSrc 是指向媒体类型结构体的常量指针，函数返回指向一个新的媒体类型

```
HRESULT WINAPI CopyMediaType(
    AM_MEDIA_TYPE *pmtTarget,
    const AM_MEDIA_TYPE *pmtSource);
```

功　能	这个函数复制一个结构到另一个结构中去。这个函数也要重新分配内存给目的结构，如果 pmtTarget 已经包含一个内存块，就要内存泄露，因此在调用该函数前后都要调用 FreeMediaType 函数，释放格式块 Format Block
参数说明	pmtTarget 是复制的目标地址，pmtSource 是复制的源地址。若复制成功则返回 S_OK，否则返回 E_OUTOFMEMORY

```
STDAPI CreateAudioMediaType(
    const WAVEFORMATEX *pwfx,
    AM_MEDIA_TYPE *pmt,
    BOOL bSetFormat);
```

功　能	该函数利用一个.WAVEFORMATIEX 结构来初始化媒体类型。如果 bSetFormat 参数为 TRUE，该函数就分配一块新的内存，如果原来的 pmt 已经包含内存，就有可能发生内存泄露。为了避免内存泄露，在调用这个函数前后都要调用 FreeMediaType，释放格式块 Format Block
参数说明	pwfx 指向 WAVEFORMATIEX 结构体，pmt 指向将要初始化的 AM_MEDIA_TYPE 结构体，bSetFormat 表示是否初始化格式块。函数返回 S_OK 或 E_OUTOFMEMORY

2）内存释放（2个函数）

```
void WINAPI FreeMediaType(
    AM_MEDIA_TYPE& mt);
```

功　能	用来释放格式块的内存
参数说明	mt 是将要释放的媒体类型结构体，函数无返回值

```
void WINAPI DeleteMediaType(
    AM_MEDIA_TYPE *pmt);
```

功　能	释放一个 AM_MEDIA_TYPE 结构
参数说明	删除由 CoTaskMemAlloc 或CreateMediaType创建的媒体类型，函数无返回值

4. 媒体样本（Media Sample）和分配器（Allocator）

滤波器通过引脚与引脚之间的连接来递交数据，数据从一个滤波器的输出引脚转移到另一个滤波器的输入引脚，除了个别情况，实现这种功能通常的方法是调用输入引脚上的 IMemInputPin::Receive 方法。该函数原型如下。

```
HRESULT Receive(
  IMediaSample *pSample  //指向IMediaSample结构体
);
```

该函数的返回值如表 7-2 所示。

表 7-2　Receive 返回值列表

返回值	意义描述
S_OK	成功
S_FALSE	样本被拒绝
E_POINTER	指针参数为空
VFW_E_INVALIDMEDIATYPE	无效的媒体类型
VFW_E_RUNTIME_ERROR	产生了运行时错误
VFW_E_WRONG_STATE	引脚停止

利用滤波器，媒体数据的内存空间可以通过多个途径来分配：在堆栈内、DirectDraw 表面（Surface）、共享 GDI 内存或使用其他的分配机制。负责分配内存空间的对象称为分配器（Allocator），是一个暴露 IMemAllocator 接口的 COM 对象。

当两个引脚相连时，其中的一个引脚必须提供一个分配器，DirectShow 定义了一个方法调用序列来决定到底由哪个引脚来提供分配器，引脚还负责协商分配器创建的 Buffer 数目和每个 Buffer 的大小。

在数据流开始之前，分配器创建了一个缓冲池（Pool of Buffer）。在数据流动过程中，上游滤波器在缓冲中填入数据并递送给下游滤波器，但是，上游滤波器递送给下游滤波器的并不是原始的缓冲区指针，而是一个称为媒体样本（Media Sample）的 COM 对象，它由分配

器创建并用来管理缓冲区，暴露 IMediaSample 接口。一个媒体样本滤波器（Media Sample）包含如下几部分。

- 指向下层缓冲区的指针。
- 时间戳。
- 各种标记。
- 可选的媒体类型。

时间戳定义了呈现时间（Presentation Time），用以 Renderer Filter 确定播放的合适时机。各种标记可以用来标识很多工作，比如，数据在上一个样本后是否被中断过（如重新定位、丢帧等）。媒体类型为流中间改变数据格式提供了途径，通常，没有媒体类型的样本，被认为从上一个样本以后的数据格式没有被改变过。

当滤波器使用一个缓冲时，它保存了样本上的参考计数。分配器使用参考计数来决定什么时候可以重用这个缓冲，从而防止了一个滤波器在写一个缓冲时另一个滤波器还在使用这个缓冲，除非所有的滤波器都释放了这个缓冲，否则样本不会将其返回给分配器的缓冲池。

5．硬件如何参与 Filter Graph

我们知道是 DirectShow 帮助应用程序把数据读写到音频、视频硬件的。那硬件又是如何参与、协调工作的呢？

1）包装滤波器（Wrapper Filter）

所有的 DirectShow Filter 都是用户模式的 COM 组件。为了使诸如视频采集卡这样的内核模式的硬件驱动加入到 Filter Graph 中，必须使其具有用户模式的滤波器特点和行为。DirectShow 提供包装滤波器来完成这个功能，这类滤波器包括 Audio Capture Filter、VFW Capture Filter、TV Tuner Filter、TV Audio Filter 和 Analog Video Crossbar Filter。DirectShow 也提供一个叫 KsProxy 的 Filter，它可以实现任何类型的 WDM 流驱动。硬件商通过提供一个 KsProxy plug-in 来扩展 KsProxy，以使其支持自己的特定功能，KsProxy plug-in 是一个被 KsProxy 聚合的 COM 对象。

包装滤波器通过暴露 COM 接口来实现特定设备的功能。应用程序使用这些接口将信息传递给滤波器，滤波器再把这些 COM 调用转化为设备驱动调用，将信息传递到内核模式下的设备中去，然后返回结果给上层应用程序。TV Tuner、TV Audio、Analog Video Crossbar 和 KsProxy Filter 都通过 IKsPropertySet 接口来支持驱动的自定义属性，VFW Capture Filter 和 Audio Capture Filter 不支持这种方式。

包装滤波器使应用程序可以像控制其他 Directshow Filter 一样来控制设备，滤波器已经封装了与内核驱动通信的细节。从而为应用层访问音/视频硬件设备极大地降低难度，开发出稳定、强大的应用程序。

2）Windows 视频设备

VFW Capture Filter 支持早期的 VFW 模式采集卡，当一个设备加入到操作系统中后，它可以被 Directshow 使用系统设备枚举器 System Device Enumerator 发现并加入到 Filter Graph 中去。VFW 采集卡逐渐被新的 WDM 模式采集卡所取代。

3）音频采集和混音设备（Mixing Device/Sound Card）

较新的声卡都有麦克风等设备的插口，而且大多数这类声卡都有硬件级的混频能力，可单独控制每一个连接设备的音量及高低音。在 **DirectShow** 中，声卡的输入和混频设备被 Audio Capture Filter 封装。每个声卡都能被系统设备枚举器发现。要查看用户系统中所有的声卡，只需打开 GraphEdit，从 Audio Capture Sources 一类中选择即可，每个在这个类里的 Filter 都是一个单独的音频捕获滤波器（Audio Capture Filter）。

4）WDM 流设备（Streaming Devices）

较新的硬解码设备和采集卡都遵照 WDM 规范。这些设备比 VFW 设备具有更强大的功能，同时可以运行于多种操作系统（Win98/ME、Win2000、WinXP）中。WDM 视频采集卡具有许多 VFW 卡所没有的功能，包括枚举采集的格式、编程调整视频图像参数（如对比度、亮度、饱和度等）、编程选择输入端和电视调谐支持。

DirectShow 为了支持 WDM 流设备，提供了 KsProxy Filter（ksproxy.ax）。KsProxy 被称为“瑞士军刀”，因为它可以做很多不同的事情。滤波器上引脚的数量，以及 COM 接口的数量，取决于底层驱动的能力。KsProxy 不以“KsProxy”这个名字显示在 Filter Graph 中，而是使用一个已在注册表中登记的设备名称。要查看用户系统中的 WDM 设备，可以运行 GraphEdit 然后从 WDM Streaming 这个类别中选择。即使用户的系统中只有一块 WDM 卡，这块卡也可能包含多个设备，而每一个设备都表现为一个滤波器，每个滤波器是实际意义上的 KsProxy。

应用程序使用“系统设备枚举器”（System Device Enumerator）在系统中寻找 WDM 设备 Moniker，然后调用 Moniker 的 BindToObject 来实例化。因为 KsProxy 能够表现所有类型的 WDM 设备，因此它必须通过询问驱动来决定哪些属性是驱动所支持的。属性集是一组数据结构的集合，被 WDM 设备使用，也被诸如 MPEG2 软解码 Filter 这样的用户模式 filter 使用。KsProxy 通过暴露 COM 接口来配置自己，硬件制造商则通过提供插件 Plug-ins 来扩展 KsProxy，插件暴露硬件商自定义的一些接口，用以实现特殊的功能，所有这些细节对于应用程序来说都是不可见的，应用程序通过 KsProxy 控制设备就像控制其他的 DirectShow Filter 一样。

5）内核流 Kernel Streaming

WDM 设备支持内核流，在内核流中数据在内核模式下被彻底流化，而永远不需要切换到用户模式下去，从而避免了在内核模式和用户模式之间切换的巨大开销，内核流允许高比特率而不消耗 CPU 的时间。基于 WDM 的 Filter 能够使用内核流将多媒体数据一个硬件设备送入到另一个中去，既可以是在同一块卡中也可以在不同的卡中，而不需要将数据复制到系统内存。

从应用程序的视点来看，数据好像是从一个用户模式的滤波器传到另一个滤波器中去，但是实际上，数据根本就没有传到用户模式下过，而是可能支接从内核模式的设备中传到下一个中去直至被呈现（Render）在显卡上。某些情况，比如采集视频到一个文件中去，在某些点上需要将数据从内核模式传入到用户模式，但是，仍然没有必要将数据复制到内存的一个新位置中去。

当然以上这些概念可能比较抽象，也确实如此，WDM 的技术本身较为复杂，编写 WDM 驱动本身就是非常有挑战性的工作。还好，对于多数的应用开发人员，也没有必要去深究这些底层的东西，DirectShow 已经为我们按照一个统一的模式——滤波器概念包装起来了。

7.3.4 构建滤波器链表（Building Filer Graph）

构建滤波器链表，包括链表构建组件、链表构建概览和智能链接。

1．链表构建组件（Graph-Building Components）

DirectShow 提供了一系列用于构建 Filter Graph 的组件。

- **Filter Graph Manager**：这个 COM 对象用于控制 Filter Graph，支持 IGraphBuilder、IMediaControl 和 IMediaEventEx 等许多接口，所有的 DirectShow 应用程序都需要在某些地方用到这个对象，虽然在有些情况下，是其他的对象为应用程序创建了 Filter Graph Manager。
- **Capture Graph Builder**：这个对象为构建 Filter Graph 提供附加方法。它最初是为构建提供视频采集的 Graph 而设计的（这正是它的名字由来），但是对于构建许多另外类型的 Filter Graph 也是很有用的，它支持 ICaptureGraphBuilder2 接口。
- **Filter Mapper 和 System Device Enumerator**：这些对象用于查找在系统中注册的或代表硬件驱动的滤波器。
- **DVD Graph Builder**：这个对象构建用以回放和导航 DVD 的 Filter Graph。它支持 IDvdGraphBuilder 接口。基于脚本的应用程序能够使用 MSWebDVD ActiveX 控件来控制 DVD 回放。
- **Video Control**：WinXP 提供这个 ActiveX 控件，用于操纵 DirectShow 中的数字和模拟电视。

2．链表构建概述（Overview of Graph Building）

创建一个 Filter Graph，从创建一个 Filter Graph Manager 实例开始。Filter Graph Manager 支持下列 Graph，构建方法如表 7-3 所示。

```
IGraphBuilder* pIGB;                     //滤波器链表管理器接口指针
HRESULT hr = CoCreateInstance(
         CLSID_FilterGraph,              //要创建的Filter的Class ID
        NULL,                            //表示Filter不被聚合
         CLSCTX_INPROC _SERVER,          //创建进程内组件对象
         IID_IGraphBuilder,              //创建Filter成功后获得的接口ID
         (void **)&pIGB);                //生成的接口对象的指针
```

表 7-3 Filter Graph Manager 支持的 Graph 构建方法

IFilterGraph::ConnectDirect	在两个引脚之间直接进行连接，如果连接失败，则返回失败
IFilterGraph::Connect	连接两个引脚，如果可能的话直接连接它们，否则，在中间加入其他的 Filter 来完成连接
IGraphBuilder::Render	从某个输出引脚处开始完成剩余的 Graph 构建。该方法会自动在输出引脚后面添加必要的 Filter，直到 Renderer Filter 成功为止
IGraphBuilder::RenderFile	构建一个完整的文件回放 Graph
IGraphBuilder::AddFilter	作用是将一个 Filter 添加到 Graph 中，它并不连接 Filter，并且在调用此方法前，Filter 必须已经被创建。创建 Filter 可以是用 CoCreateInstance 方法或使用 Filter Mapper 或系统设备枚举器 System Device Enumerator

这些方法提供了 3 种构建 Graph 的途径。

- Filter Graph Manager 构建整个 Graph。
- Filter Graph Manager 构建部分 Graph。
- 应用程序构建整个 Graph。

1）Filter Graph Manager 构建整个 Graph

如果用户仅仅是想回放一个已知格式的文件，如 AVI、MPEG、WAV 或 MP3，则使用 RenderFile 方法。

RenderFile 方法首先寻找注册在系统中能分析源文件的 Filter，它使用协议名（如文件名中的 http://）、文件扩展名或文件的头几个字节来决定选择哪一个源滤波器。

滤波器链表管理器使用一个迭代过程来完成余下的 Graph 构建。在这个迭代过程中，它逐个列出 Filter 的输出引脚上支持的媒体类型，并搜索哪个已注册的滤波器的输入引脚接受该媒体类型。它使用一系列的规则来缩小滤波器的范围并排定优先顺序。

- 滤波器类别（Filter Category）标识的滤波器的一般功能。
- 媒体类型描述 Filter 能在接受输入或能输出的数据类型。
- 特征值 Merit 决定 Filter 被尝试的顺序。如果两个滤波器具有相同的滤波器类别，并且同时支持相同的输入类型，则 Filter Graph Manager 选择 Merit 值较大的那一个。一些滤波器故意给出一个小 Merit 值是因为它是为特殊用途而设计的，仅能由应用程序来将其添加到 Graph。

Filter Graph Manager 使用 Filter Mapper 对象来搜索已注册的滤波器。

每个滤波器被添加时，Filter Graph Manager 试着将其与前一个滤波器的输出引脚连接。它们协商决定引脚之间是否能连接，如果能，则哪一种媒体类型又被用来连接。如果新滤波器不能连接，则 Filter Graph Manager 丢弃它并尝试另一个，这个过程一直持续到每个流都被呈现为止。

2）Filter Graph Manager 构建部分 Graph

如果我们不仅仅是播放一个文件，那么应用程序就必须做一些 Graph 的构建工作。比如，一个视频采集应用程序必须先选择一个 Source Filter 并将其添加到 Graph 中去。如果用户需要将数据写入到一个 AVI 文件中，就必须添加一个 AVI Mux 和 File Write Filter。不过，也经

常有可能让 Filter Graph Manager 自己来完成整个 Graph 的构建，比如，用户可以通过 Render 方法来呈现一个引脚实现预览。

3）应用程序构建整个 Graph

在某些场合，我们的应用程序可能会需要添加、连接每个滤波器来构建 Graph。在这种情况下，用户很可能明确地知道哪些滤波器需要加到 Graph 中去。使用这种方式，应用程序通过调用 AddFilter 方法添加每个滤波器，然后枚举滤波器上的引脚，调用 Connect 或 ConnectDirect 来连接它们。

3．智能链接（Intelligent Connect）

智能链接覆盖了一系列滤波器链表管理器（Filter Graph Manager），用于构建所有或部分 Filter Graph 的方法。在任何时候，当 Filter Graph Manager 需要添加 Filter 来完成 Graph 时，它大致做以下几件事情。

- 如果有一个 Filter 存在于 Graph 中，并且该 Filter 至少有一个没有连接的 Input 引脚，则 Filter Graph Manager 试着去试用这个 Filter。否则，Filter Graph Manager 在已注册的 Filter 中寻找连接时可以接受合适的媒体类型的 Filter。每一个 Filter 都注册有一个特征值 Merit，这个值用以标记哪个 Filter 最容易被 Filter Graph Manager 选中来完成 Graph。Filter Graph Manager 按 Merit 值的顺序来选择 Filter，Merit 值越大，被选中的机会就越大。对于每种流类型（如音频、视频、MIDI），默认的 Renderer 具有一个很高的 Merit 值，解码器同样如此，专用 Filter 具有低 Merit 值。
- 如果 Filter Graph Manager 因选择的 Filter 不合适而被拒绝，则它会返回来尝试另外的 filter 组合。智能链接是 Filter Graph Manager 用以构建 Filter Graph 的机制。它包含了一系列的相关的用于选择滤波器及将其添加到 Graph 中的算法。其实，作为应用程序开发者，并不需要很具体地了解智能连接的细节。如果在构建某个 Filter Graph 时遇到问题并希望能解决它，或者用户正在编写自己的滤波器并希望它能自动地被 Graph 构建，这一节很有帮助。

智能链接涉及以下 IGraphBuilder 方法，GraphBuilder::Render 的原型：

```
HRESULT Render(
        IPin *ppinOut);
```

ppinOut 指向输出的引脚，函数返回值如表 7-4 所示。

表 7-4　Render 方法的返回值及意义描述

返回值	意义描述
S_OK	成功
VFW_S_AUDIO_NOT_RENDERED	部分成功；语音没有被渲染成功
VFW_S_DUPLICATE_NAME	成功；链表管理器已经修改了滤波器的名字而避免复制
VFW_S_PARTIAL_RENDER	部分成功；该影音中某些流是不支持的格式
VFW_S_VIDEO_NOT_RENDERED	部分成功；视频没有被渲染
E_ABORT	操作终止

（续表）

返回值	意义描述
E_OUTOFMEMORY	内存不足
E_POINTER	指针为空 NULL
VFW_E_CANNOT_CONNECT	滤波器无法链接
VFW_E_CANNOT_RENDER	滤波器无法渲染流媒体
VFW_E_NO_ACCEPTABLE_TYPES	在引脚间没有公共的媒体格式
VFW_E_NOT_IN_GRAPH	引脚所属的滤波器不在滤波器图表中

Render 方法构建一部分 Graph，它从一个尚未连接的输出引脚开始顺着数据流的方向往下，添加必要的滤波器，但起始的那个滤波器必须已被添加到 Graph 中了。Render 方法每一步都搜索一个能够连接到前一个 Filter 的 Filter，如果新连接上的 Filter 有多个输出引脚，则数据流自动分流搜索，直到每个流都被 Renderer 为止。如果 Render 方法搜索到的滤波器无法使用，它会返回去尝试另外一个滤波器。要链接每一个输出引脚，Render 方法做以下工作。

- 如果引脚支持 IStreamBuilder 接口，则 Filter Graph Manager 让引脚的 Istream Builder::Render 方法来完成整个过程。通过暴露这个接口，引脚承担了构建 Graph 剩余部分的全部工作。但是，只有很少数的 Filter 支持此接口。
- Filter Graph Manager 尝试使用任何在缓存中的 Filter。在智能链接的整个过程中，Filter Graph Manager 可以在早期将 Filter 缓存起来。
- 如果 Filter Graph 包含了任何有未连接的输入引脚的滤波器，Filter Graph Manager 会将其当做下一个滤波器来尝试连接。用户可以通过在调用 Render 之前添加特定的 Filter 来强制让 Render 方法尝试这个滤波器。
- 最后，Filter Graph Manager 使用 IFilterMapper2::EnumMatchingFilters 方法在所有注册的 Filter 中寻找，依据已注册的媒体类型列表，逐个试着匹配输出引脚的各个媒体类型（按优先级高低排列）。

每个已注册的滤波器都有一个 Merit 值，这是用来表示滤波器优先级的数字，最大优先级越高，EnumMatchingFilters 方法返回的滤波器集依据 Merit 值来排列，直至最小的 Merit 值 MERIT_DO_NOT_USE+1，它忽略 Merit 为 MERIT_DO_NOT_USR 或更小的滤波器。滤波器也通过 GUID 来归类，类别本身也有 Merit 值，EnumMatchingFilters 方法忽略任何 Merit 值为 MERIT_DO_NOT_USE 或更小的类别，即使在那个类别中的滤波器有较高的 Merit 值。总的来说，Render 方法以下列步骤尝试 Filter。

STEP 01 使用 IstreamBuilder。

STEP 02 尝试被缓存的 Filter。

STEP 03 尝试已添加在 Graph 中的 Filter。

STEP 04 在已注册的 Filter 中寻找。

IGraphBuilder::AddSourceFilter 的原型如下。

```
HRESULT AddSourceFilter(
            LPCWSTR lpwstrFileName,
```

```
                LPCWSTR lpwstrFilterName,
                IBaseFilter **ppFilter);
```

lpwstrFileName 表示预加载的文件名，lpwstrFilterName 表示源滤波器名，ppFilter 指向IBaseFilter接口的源滤波器，该函数返回值如表 7-5 所示。

表 7-5 返回值及意义描述

返回值	意义描述
S_OK	成功
E_NOINTERFACE	源滤波器不支持IFileSourceFilter接口
E_OUTOFMEMORY	内存不足
E_POINTER	指针为空
VFW_E_CANNOT_LOAD_SOURCE_FILTER	该媒体文件的源滤波器没有被加载
VFW_E_NOT_FOUND	没有找到文件或目标
VFW_E_UNKNOWN_FILE_TYPE	该媒体文件格式不支持

AddSourceFilter 方法添加一个能 Render 特定文件的 Source Filter。首先，它依据协议名（如 Http://）、文件扩展名或文件的头几个字节在已注册的 Filter 中寻找匹配的那个。如果此方法定位到了一个合适的 Source Filter，它便立刻创建一个这个滤波器的实例，并将其添加到 Graph 中，然后调用滤波器的 IFileSourceFilter::Load 方法，参数就是文件名。

IGraphBuilder::RenderFile 的原型如下：

```
HRESULT RenderFile(
        LPCWSTR lpwstrFile,
        LPCWSTR lpwstrPlayList);
```

lpwstrFile 指向宽字符的媒体文件名，lpwstrPlayList 为保留值，设置为 NULL。函数返回值请参考上述其他函数或 MSDN。

RenderFile 方法依据一个文件名来构建一个默认的回放 Graph，在其内部，RenderFile 方法调用 AddSourceFilter 来定位 Source Filter，并且用 Render 来构建 Graph 的余下部分。

IGraphBuilder::Connect 的原型如下。

```
HRESULT Connect(
            IPin *ppinOut,
            IPin *ppinIn);
```

ppinOut 指向输出引脚，ppinIn 指向输入引脚。函数返回值请参考上述其他函数或 MSDN。

Connect 方法将输出引脚连接到输入引脚上去，这个方法自动添加必要的中间 Filter 到 Graph 中去，使用在 Render 方法中描述的那一系列方法。

- 尝试在 Filter 间直接连接。
- 尝试被缓存的 Filter。
- 尝试已添加在 Graph 中的 Filter。
- 在已注册的 Filter 中寻找。

7.3.5 滤波器链表中的数据流动Data Flow

本节主要描述在滤波器链表中媒体数据如何传送。在编写 DirectShow 应用程序时，其实不必了解这些细节，当然掌握了细节在某些情况下是有用的。在编写自己的 Filter 时，这部分是必须要掌握的。

包括传输（Transports）、样本和分配器（Samples and Allocators）、滤波器状态和拉模式（Pull Model）。

1．传输（Transports）

为了能在滤波器链表中传送、移动媒体数据，DirectShow 滤波器必须支持一些协议。这些协议被称为传输（Transport）。当两个滤波器相连时，它们必须支持相同的传输协议，否则不能交换媒体数据。典型的是，一个协议要求许多引脚中的一个支持特定的接口。当滤波器之间相连时，引脚用这个接口询问其他接口。

多数的 DirectShow 滤波器都把媒体数据保存在主存储器中，并通过引脚把数据提交给其他的滤波器，这种传输称为局部存储器传输（Local Memory Transport）。虽然局部存储器传输在 DirectShow 中最常用，但并不是所有的滤波器都使用它。例如，有些滤波器通过硬件传送媒体数据，引脚只是用来提交控制信息，如 IOverlay 接口。

DirectShow 为局部存储器传输定义了两种机制：推模式（Push Model）和拉模式（Pull Model)。在推模式中，源过滤器生成数据并提交给下一级滤波器，下一级过滤器被动地接收数据，完成处理后再传送给再下一级过滤器。在拉模式中，源滤波器与一个分析滤波器相连，分析滤波器向源滤波器发送请求数据后，源滤波器才传送数据以响应请求。推模式使用的是 IMemInputPin 接口，拉模式使用 IAsyncReader 接口，而推模式比拉模式要更为常用。后面的内容中，给出了拉模式使用 IAsyncReader 接口的应用过程。

2．样本（Samples）和分配器（Allocators）

在前面 **“媒体样本和分配器”** 中提到过样本和分配器，这里再详细地展开描述。当一个引脚向另一个引脚传递数据的时候，它并不是直接将内存块的指针传递下给一个引脚，实际上，它将传递一个管理内存的 COM 对象的指针给下一个引脚。这个 COM 对象称为媒体样本，它暴露了 IMediaSample 接口。接收引脚通过调用 IMediaSample 的方法来对内存进行操作，比如方法 IMediaSample::GetSize,IMediaSample::GetActualDataLength 及 IMediaSample::GetPointer。

样本一般都是从源 Filter 开始，通过输出引脚传递到下一个 Filter 的输入引脚，一直传递下去直到 Render Filter。在拉模式中，输出引脚通过调用输入引脚上的 IMemInputPin::Receive 方法传递样本，输入引脚或者在 Receive 函数同步处理数据，或者采用另外一个工作线程异步处理的方式。如果输入引脚在 Recive 方法中需要等待资源，也可以阻塞等待。

另外一个 COM 对象叫做分配器，用来创建和管理样本，暴露了 IMemAllocator 接口。当一个 Filter 需要一个空的 Buffer 的时候，就可以调用 IMemAllocator::GetBuffer 方法，该方法返回一个指向样本的指针。每一个引脚连接都共享一个分配器，当两个引脚连接的时候，它们会决定由哪个 Filter 来提供分配器。通过引脚还可以设置分配器的属性，例如 Buffer 的

数量和大小。

图 7-31 显示了 Sample、Allocator 和 Filter 之间的关系。滤波器的输入引脚和输出引脚均调用 Allocator 方法申请内存，然后对各个媒体文件申请缓冲区 Buffer。

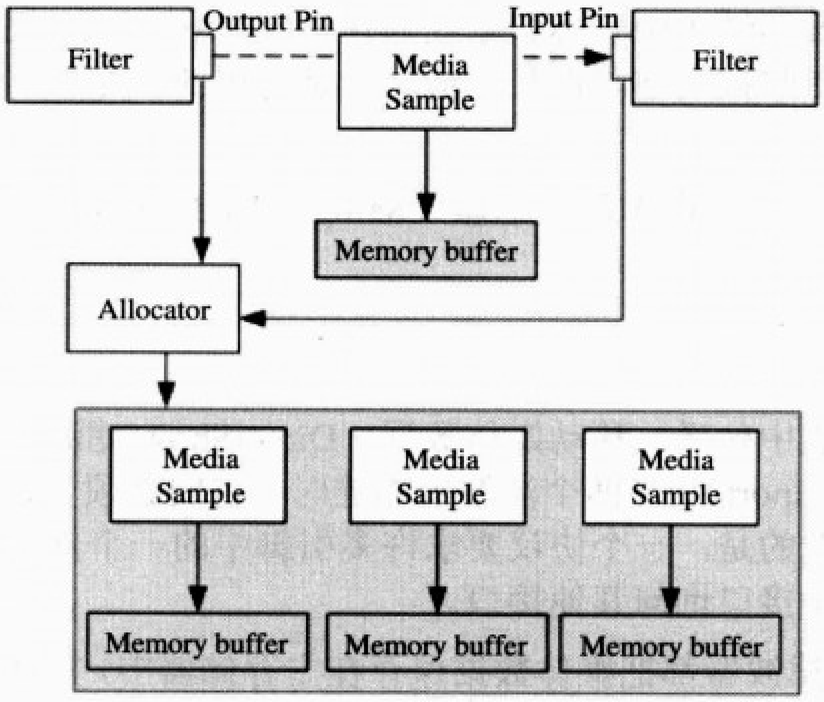


图 7-31　样本、分配器和滤波器之间的关系框图

1）媒体样本引用计数

分配器创建一个有限的媒体样本池，在任何时候，有些样本被使用，其他部分闲置而等待被 GetBuffer 调用。分配器通过引用计数来跟踪样本，滤波器调用 GetBuffer 方法返回引用数为 1 的媒体样本。如果样本的引用计数为 0，样本就返回内存池，成为空闲的样本，可以再次被 GetBuffer 调用。主要引用计数大于 0，则没有样本可被 GetBuffer 调用。如果分配器的每个样本都在被使用，则 GetBuffer 会被阻塞，直到有一个样本空闲。

比如，假设一个输入引脚接到一个样本，如果它在 Receive 方法里同步地处理这个样本，而没有增加该样本的引用计数，则在 Receive 返回后，输出引脚就释放这个样本，引用计数变为 0，样本就返回到内存池中。换句话说，如果输入引脚在一个工作线程内处理样本，它会在离开 Receive 方法之前增加引用计数，引用计数就是 2。当输出引脚释放样本时，计数变为 1，样本还没有返回到样本池中。在样本工作的线程处理完时，调用 Release 释放样本，样本返回样本池。

当一个引脚接收一个样本时，它可以将数据复制到另一个样本中，也能修改原来的样本，同时把那个样本传递到下一个 Filter。更有甚者，一个样本可以流遍整个 Filter Graph，每个滤波器依次调用 AddRef 和 Release 方法。所以，当一个输出引脚调用了 Release 后，就不应该再次使用该样本，因为也许下游的 Filter 正在使用它。输出引脚必须总是调用 GetBuffer 方法获取新的样本。

这种机制减少了内存分配的数量，因为 Buffer 可以被重复使用。也防止了数据没有被处理的样本被意外地写入，因为分配器维护着一个样本可用的列表。

一个滤波器可以使用分离的分配器来作为输入和输出，在扩展输入数据时（如解码），可能就是如此。如果输出不会超过输入，则滤波器可能会就地处理数据而没有复制到新的样本中，在这种情况下，两个或更多的引脚连接可以共享一个分配器。

2）分配和释放内存

当一个 Filter 创建一个分配器时，分配器还没有保留任何的内存缓冲区，这个时候任何程序调用 Getbuffer 方法都会失败。只有当数据流开始的时候，输出引脚调用 IMemAllocator::Commit 方法，提交分配器，才分配分配器内存，此时引脚可以调用 GetBuffer 方法。

当数据流停止时，引脚则调用 IMemAllocator::Decommit 方法来销毁分配器。在分配器再次分配之前，所有调用 GetBuffer 方法都会失败。当然，如果任何 GetBuffer 的调用被阻塞而在等待一个样本，则会立即返回一个错误码。根据执行的情况，Decommit 方法释放内存有可能成功有可能失败。比如，CMemAllocator 类等待直到它的析构器方法释放内存。

3. 滤波器（Filter）状态

滤波器有 3 种可能的状态：停止、暂停和运行。暂停状态的目的是在链表中插入数据，使能立刻响应运行命令。滤波器链表管理器控制着滤波器的所有状态的转换。当应用程序调用 IMediaControl::Run、IMediaControl::Pause 或者 IMediaControl::Stop 时，过滤器图表管理器就调用滤波器相应的 IMediaFilter 方法。停止、运行状态的切换总是要经过暂停的，因此，当一个应用程序对一个停止的 Graph 调用 Run 命令时，过滤器图表管理器在运行之前首先要暂停。

对于大多数的滤波器来说，Running 和 Paused 状态是一样的。看下面典型的 Graph：

```
Source > Transform > Renderer
```

目前一般都认为源滤波器（Source Filter）不是一个实时捕获源滤波器。当源滤波器暂停时，它会创建一个线程，用来产生新的数据并以尽可能快的速度写入到媒体样本中。这个线程通过调用转换滤波器（Transform Filter）的输入引脚上的 IMemInputPin::Receive 方法向下游“推送”样本。转换滤波器在滤波器的线程内接收样本，同时，它可能会使用一个工作线程来传送样本到 Renderer，但更多情况是在同一个线程内。当 Renderer 暂停时，它等待接收一个样本，在它接收到一个后，Renderer 阻塞并保留那个不确定的样本。如果这个 Renderer 是一个视频 Renderer，它会作为一个广告图像来显示，有必要的话会重新绘制。

基于以上工作，媒体的样本流被完全插入并做好渲染的准备。如果链表保持暂停，样本就被堆积在第一个样本之后的链表中，直到每个滤波器在 Receive 或 GetBuffer 方法中被阻塞，这样就不会有数据丢失。一旦源线程解锁，它也只会从阻塞的那一点重启。

源滤波器和转换滤波器会忽略从“暂停”到“运行”的状态切换，这样它们只是尽可能快地连续处理数据。但是当渲染器 Renderer 运行时，它会渲染样本。首先，Renderder 渲染在样本暂停时被保留的那些样本，然后每次接收新的样本，并计算样本的呈现时间 Presentation Time。Renderer 保留每个样本直到被呈现的时间，也就是在那一点上 Renderder 渲染样本。在 Renderer 等待这个呈现时间的同时，它或者阻塞在 Receive 方法中，或者在一个线程序列的工作线程中接收新的样本。从 Renderer 回溯上游的滤波器没有被涉及到这个调度任务中。

现场信源（Live Source），如捕获设备对于这个普通的架构来说却是一个例外。因为它不可能预先插入任何数据。这种应用可能使得链表暂停，且在运行前链表会有段长时间的等待。链表不会渲染旧的样本，所以在暂停时，现场信源不会产生样本，只有在运行时才会。为了把这个实事通知给滤波器链表管理器，源滤波器的 IMediaFilter::GetState 方法会返回

VFW_S_CANT_CUE 标识，这个返回码表明即使 Renderer 没有收到任何数据，滤波器也已经切换到暂停状态。

当一个滤波器停止时，它拒绝发送给它的任何样本。源滤波器关闭它们的线程，其他滤波器也关闭它们创建的有关线程，引脚释放内存分配器分配的内存。

滤波器链表管理器按照逆流的方向来实现 Filter 的状态切换，从末端的 Renderer 滤波器回溯直到源滤波器。这种方式对防止样本丢失、防止链表死锁是必需的。最关键的状态切换是在暂停和停止之间。

- 停止到暂停。当滤波器暂停时，它就做好了从下一个滤波器接收样本的准备，源滤波器是最后一个切换到暂停的。它开始创建线程，发送滤波器，因为下游的滤波器的状态都已经切换到暂停了，所以，所有的滤波器都不会拒绝而是接收样本，只有当所有的渲染器 Renderer 都接收到这个样本时，滤波器链表管理器才算完成了状态的切换。
- 暂停到停止。当一个滤波器停止时，它要释放它拥有的所有的样本，同时它不阻塞在 GetBuffer 中等待的上游滤波器。如果这个滤波器正在等待 Receive 方法的内部资源，它停止等待而从 Receive 方法中返回，从而不阻塞那个调用滤波器，所以，当滤波器链表管理器试图停止上游的一个滤波器时，这个滤波器不会阻塞在 Getbuffer 和 Receive 方法里，它会立即响应 Stop 命令。上游的 Filter 也许在执行 Stop 命令前还会将少量的样本传递下去，但是下游的滤波器只会拒绝，因为它们已经停止工作了。

4．拉模式（Pull Model）

在 IMemInputPin 接口中，上游的滤波器决定发送什么样的数据，然后把数据推给下游的滤波器。而对于某些滤波器，拉模式可能更合适。下游的滤波器向上游的滤波器请求数据，媒体数据依然是从上游到下游，从输出引脚到输入引脚，但是，是下游的滤波器发起数据的流动动作。这种类型的连接使用的是 IAsyncReader 接口。

拉模式的典型应用是在文件的回放中，例如在一个 AVI 文件的回放 Graph 中，异步的源文件滤波器（Async File Source Filter）就担负着从文件中读取数据的一般任务，然后将数据以字节流、无格式信息的的方式发送给下面的滤波器。

输入引脚从输出引脚中请求数据，使用的方法有 IAsyncReader::Request、Iasync Reader::SyncRead 和 IAsyncReader::SyncReadAligned。第一个方法是异步的，支持多个重复的读，另外两个是同步工作的。函数原型如下。

```
HRESULT Request(
  IMediaSample *pSample,
  DWORD_PTR dwUser
);
```

pSample 指向IMediaSample接口类型的媒体样本，dwUser 表示当请求完成时返回的用户特定值。函数原型如下。

```
HRESULT SyncRead(
  LONGLONG llPosition,
  LONG lLength,
```

```
  BYTE *pBuffer
);
```

llPosition 说明开始读的字节偏移量，如果超过了文件末尾则该方法失败，lLength 表示读取的长度字节数，pBuffer 接收数据的缓冲区。该函数是采用同步方式读取的，函数原型如下。

```
HRESULT SyncReadAligned(
  IMediaSample *pSample
);
```

pSample 指向 IMediaSample 接口类型的媒体样本。该函数同样是同步方式读取，只不过是字节对其方式读取。

从理论上讲，任何滤波器都支持接口 IAsyncReader，而实际上，仅仅是那些连接到剖析滤波器的源滤波器才支持。而剖析滤波器的工作方式非常类似于在推模式中的源滤波器。当剖析器暂停时，它会产生一个流线程从 IAsyncReader 连接来拉数据，进而推送给下游。

7.3.6 DirectShow中的事件通知机制

在 Windows 编程时，经常使用消息、消息队列和消息响应等概念，而 DirectShow 中的事件通知机制和 Windows 中的消息机制非常类似。各种滤波器、链表管理器等都是通过事件通知的方式和应用程序交互、通信的。比如，媒体的运行、暂停或停止状态，滤波器发送相应的事件给应用程序，以便我们做相关的处理，如打开文件、显示或重启播放等。

1. 事件通知机制概述

当某个事件发生时，比如数据流结束，产生一个错误等，滤波器就给滤波器图表管理器发送一个事件通知。滤波器图表管理器处理其中的一部分事件，另一部分交给应用程序处理。如果图表管理器没有处理一个滤波器事件，它就把事件通知放入到一个队列中，图表管理器也可以将自己的事件通知放进队列中。

滤波器通过发送一个事件通知给滤波器链表管理器表明一个事件发生，事件可以使数据流结束，产生了数个错误（如 Render 流失败等）。滤波器链表管理器自己处理一部分事件，其他事件提交给应用程序处理。如果链表管理器没有处理一个滤波器事件，则把事件通知放入一个队列中。并且，链表管理器也可以把自己的事件通知放入队列中。

应用程序从队列中提取事件，并根据事件类型作出反应。因此，DirectShow 的事件通知非常类似于 Windows 中的消息队列机制。对于给定的事件类型，应用程序也能够取消滤波器链表管理器的默认行为，那么，滤波器链表管理器就把那些事件直接放入队列中，让应用程序处理。这种事件机制能够使得：

- 滤波器链表管理器和应用层通信；
- 滤波器既与应用层又与滤波器链表管理器通信；
- 应用程序在处理事件过程中决定干预的程度大小。

2. 检索事件

滤波器链表管理器暴露了 3 个接口用来处理事件通知。

- IMediaEventSink：滤波器用这个接口来发布事件。
- IMediaEvent：应用程序用这个接口来从队列中检索事件。
- IMediaEventEx：Imediaevent 的继承和扩展。

滤波器都是通过调用链表管理器的 IMediaEventSink::Notify 方法来通知图表管理器某种事件的发生的。事件通知可包括一个事件码，这个代码不仅代表了事件类型，还包含两个 DWORD 类型的参数，用来传递一些补充信息。依赖于事件码的这些参数可能包含指针、返回码引用次数或者其他信息。事件码和参数的列表，详见“事件通知码”。

应用程序通过调用图表管理器的 IMediaEvent::GetEvent 方法来从事件队列中获取事件。如果有事件发生，则该方法就返回一个事件码和两个事件参数，如果没有事件，则一直阻塞到有事件发生和超过某个设定时间。调用 GetEvent 方法后，应用程序必须调用 IMediaEvent::FreeEventParams 来释放事件码所带参数的所有资源。比如，某个参数可能是由滤波器链表分配的内存，函数原型：

```
HRESULT GetEvent(
  long *lEventCode,
  long *lParam1,
  long *lParam2,
  long msTimeout
);
```

lEventCode 指向接收事件码的变量，lParam1 指向接收第一个事件参数的变量，lParam2 指向接收第二个事件参数的变量，msTimeout 超时时间间隔，单位为 ms。用 INFINITE 来阻塞消息产生直到有事件发生，函数原型：

```
HRESULT FreeEventParams(
  long lEventCode,
  long lParam1,
  long lParam2
);
```

lEventCode 表示事件码，lParam1 表示第一个事件参数，lParam2 表示第二个事件参数。

下面的代码演示了如何从事件队列中检索事件。

```
long evCode, param1, param2;       //分别表示事件码，第一个事件参数，第二个事件参数。
HRESULT hr;                        //获取事件函数的返回值
while (hr = pEvent->GetEvent(&evCode, &param1, &param2, 0), SUCCEEDED(hr))
{
    switch(evCode)
    {
        //调用应用层预定义的函数
        //处理每种你预处理的各种类型的事件
    }
    hr = pEvent->FreeEventParams(evCode, param1, param2);
}
```

为了重载滤波器链表管理器对事件的默认处理，可以使用某个事件码作为参数来调用 IMediaEvent::CancelDefaultHandling，这样就可以屏蔽图表管理器对某个事件码的默认处理。

如果要恢复图表管理器对该事件码的默认处理，则可以调用 IMediaEvent::RestoreDefaultHandling。而如果图表管理器对某个事件码没有默认的处理，则调用这两个函数是不起作用的。

3. 分析事件

为了处理 DirectShow 事件，应用程序需要一种方法来获取正在队列中等待的事件。对此，链表管理器提供了两种方法。

- 窗口通知，链表管理器发送开发者自己预定义的窗口消息到应用程序窗口。
- 事件信号，如果队列中有 DirectShow 事件，链表管理器就用事件信号通知应用程序，如果队列为空就重新复位事件信号。

在应用程序中，这两种方法都可以采用，不过窗口通知简单些。

1）窗口通知

为设置窗口通知，调用方法 IMediaEventEx::SetNotifyWindow，并说明一个私有消息。应用程序使用从 WM_APP 开始长度为 0XBFFF 范围的消息数字作为私有信息。无论何时，链表管理器放置一个新的事件通知到队列中，并向特定的窗口发送消息。应用程序从窗口消息队列中提取消息并作出响应。

下面的代码演示了怎样设置窗口通知。

```
#define WM_GRAPHNOTIFY WM_APP + 1             //私有消息
pEvent->SetNotifyWindow((OAHWND)g_hwnd, WM_GRAPHNOTIFY, 0);
```

这个消息是个普通的 Windows 消息，并从 DirectShow 事件通知队列中发布出来。这种方法的好处是多数应用已经执行了一个消息循环，因此，用户可以不用做额外的工作就能够合并 DirectShow 事件处理。下面的例子演示了怎样响应通知消息的大体框架。

```
LRESULT CALLBACK WindowProc( HWND hwnd, UINT msg, UINT wParam, LONG lParam)
{
   switch (msg)
   {
      case WM_GRAPHNOTIFY:
         HandleEvent();  //应用层预定义的函数
         break;
      //处理Windows其他消息
   }
   return (DefWindowProc(hwnd, msg, wParam, lParam));
}
```

因为事件通知和消息循环都是异步的，所以在用户的应用程序响应消息的时候，消息队列中会包含多于一个的事件。并且，有时如果成了无效的，则事件就会被从队列中清除。所以，在用户的事件处理代码中，应该一直调用 GetEvent 方法直到返回一个错误码而表明队列已经没有消息。

在释放 IMediaEventEx 指针前，用户应通过用带空指针的 SetNotifyWindow 方法调用来取消事件通知。在用户的事件处理代码中，在调用 GetEvent 前，应检查 IMediaEventEx 指针是否为空。这些步骤都是保证在用户的应用程序中，在已释放 IMediaEventEx 指针后，不会

有可能的错误发生。

2）事件信号

链表管理器保留一个用来反映事件队列状态的手工置位事件。如果队列中含有一个挂起的事件通知，链表管理器就会发出一个手工置位事件。如果队列为空，则调用IMediaEvent::GetEvent方法置位事件。应用程序可以用这个事件来决定队列的状态。

特别要提醒的是，这里的术语可能会使我们混淆，手工置位事件是一种被Windows的CreateEvent函数产生的事件，并且对由DirectShow定义的事件不做任何事情。

调用IMediaEvent::GetEventHandle方法来获取一个指向手工置位事件的句柄。等待这个事件被诸如WaitForMultipleObjects调用而发出信号，一旦这个事件被发出信号，则调用IMediaEvent::GetEvent获取DirectShow的其他事件。

下面的例程说明了这个过程。首先程序获取事件句柄，接着为事件发出信号而等待100毫秒间隔。如果事件已经发出，则调用GetEvent方法并打印事件码及事件参数到控制窗口。当EC_COMPLETE事件发生时，循环终止，表明回放完成。

```
HANDLE  hEvent;                         //事件句柄
LONG   evCode, param1, param2;          //事件码和事件的第一、第二个参数
BOOL  bDone = FALSE;                    //已经获取到事件
HRESULT hr = S_OK;                      //获取事件句柄函数的返回值
hr = pEvent->GetEventHandle((OAEVENT*)&hEvent);//获取事件句柄
if (FAILED(hr)
{
    /*插入处理失败的代码. */
}
while(!bDone)
{
    if (WAIT_OBJECT_0 == WaitForSingleObject(hEvent, 100))
    {
/*获取事件码和第一、第二个参数，如果成功，则输出其值，然后释放事件参数*/
while (hr = pEvent->GetEvent(&evCode, &param1, &param2, 0), SUCCEEDED(hr))
{
  printf("Event code: %#04x\n Params: %d, %d\n", evCode, param1, param2);
  pEvent->FreeEventParams(evCode, param1, param2);
  bDone = (EC_COMPLETE == evCode);  //回放结束
}
    }
}
```

因为链表管理器在适当的时间会自动地设置和置位事件，所以用户的应用程序不应该这样做。当然，当用户释放链表管理器的时候，链表管理器也会关闭事件句柄，所以在那时也不要再使用事件句柄了。

4．事件通知码

现在我们来详细地描述DirectShow的事件通知码，但不包含DVD有关事件。滤波器使用IMediaEventSink::Notify方法调用来向链表管理器发送事件。链表管理器处理一部分事件，

而队列中的其他部分由应用程序来处理。应用程序通过调用 IMediaEvent::GetEvent 方法来获取、分析消息。函数原型：

```
HRESULT Notify(
  long EventCode,
  LONG_PTR EventParam1,
  LONG_PTR EventParam2
);
```

EventCode 表示事件码，EventParam1 表示第一个事件参数，EventParam2 表示第二个事件参数。

在表 7-6 中，每个条目列出了事件码、事件参数的意义，以及链表管理器的事件默认行为。使用 IMediaEvent::CancelDefaultHandling 重载这些行为，事件码在头文件 Evcode.h 和 Audevcod.h 中有定义。如果没有事件默认行为，链表管理器就会自动把事件从事件队列中推向应用程序。

滤波器可以在以 EC_USER 为开始到更大的数值来定义用户事件码。链表管理器把这些直接放入事件队列中，但是下面两点需特别注意。

- 使用通常的 IMediaEvent::FreeEventParams 方法，链表管理器不能释放事件参数。应用程序必须释放与事件参数有关的内存或引用计数等资源。
- 滤波器仅仅从准备处理事件的应用程序的内部发送事件（很有可能应用程序能在滤波器中设置用户属性来表明应用程序是安全的，可以来发送事件）。

表 7-6　DirectShow 事件通知码

事件通知码	描述
EC_ACTIVATE	视频窗口是否是活动的
EC_BUFFERING_DATA	链表正缓冲或已经停止缓冲数据
EC_BUILT	当链表已构建时视频控制发送的消息，应用层不可见
EC_CLOCK_CHANGED	参考块已经被修改
EC_CODECAPI_EVENT	编码器发送的正在编码
EC_COMPLETE	所有数据已经渲染
EC_DEVICE_LOST	即插即用设备已经删除或又已可用
EC_DISPLAY_CHANGED	显示模式已经被修改
EC_END_OF_SEGMENT	已经到达段的末尾
EC_ERROR_STILLPLAYING	异步运行链表的命令失败
EC_ERRORABORT	由于一个错误操作终止
EC_EXTDEVICE_MODE_CHANGE	不支持
EC_FULLSCREEN_LOST	渲染器全屏模式切换
EC_GRAPH_CHANGED	滤波器链表已经修改
EC_LENGTH_CHANGED	源长度已经改变

（续表）

事件通知码	描述
EC_NEED_RESTART	滤波器正在请求链表重启
EC_NOTIFY_WINDOW	通知给滤波器关于视频渲染的窗口
EC_OLE_EVENT	滤波器正在传送一个文本字符给应用程序
EC_OPENING_FILE	链表正在打开或已完成打开文件
EC_PALETTE_CHANGED	视频调色板已经被修改
EC_PAUSED	暂停请求已经完成
EC_QUALITY_CHANGE	链表正在丢样本，便于质量控制
EC_REPAINT	视频渲染器请求重绘
EC_SEGMENT_STARTED	到达一个新的段
EC_SHUTTING_DOWN	滤波器链表正在关闭，须在销毁前
EC_SNDDEV_IN_ERROR	在音频设备的输入引脚上发生了一个错误
EC_SNDDEV_OUT_ERROR	在音频设备的输出引脚上发生了一个错误
EC_STARVATION	滤波器没有收到足够的数据
EC_STATE_CHANGE	滤波器链表已经改变状态
EC_STEP_COMPLETE	滤波器执行帧步进已经完成设定的帧数
EC_STREAM_CONTROL_STARTED	一个流控开始命令已经生效
EC_STREAM_CONTROL_STOPPED	一个流控停止命令已经生效
EC_STREAM_ERROR_STILLPLAYING	流中发生了一个错误，流仍然在播放
EC_STREAM_ERROR_STOPPED	由于一个错误流已经停止
EC_TIMECODE_AVAILABLE	不支持
EC_UNBUILT	当链表已拆卸时视频控制发送的消息，应用层不可见
EC_USERABORT	用户已终止回放
EC_VIDEO_SIZE_CHANGED	视频大小已经改变
EC_VMR_RENDERDEVICE_SET	当 VMR 已选择好渲染机制发送的消息
EC_VMR_SURFACE_FLIPPED	在 VMR-7 的分配呈现器在正呈现的表面上已经调用了 Flip 方法时，而发出的消息
EC_VMR_RECONNECTION_FAILED	在不能接受来自上游解码器的动态格式变化请求时，VMR-7 和 VMR-9 发送的消息
EC_WINDOW_DESTROYED	视频渲染器已经销毁或者已经从链表中删除
EC_WMT_EVENT	在应用程序采用 ASF 读滤波器回放被 DRM 保护的 ASF 文件时，Windows 媒体格式 SDK 发送的消息

7.3.7 常用DirectShow SDK接口

DirectShow SDK 中的类或 COM 对象又称为接口（Interface），这些类或接口的成员函数用来完成流媒体数据的采集、回放和显示等动作，使用过程大体为初始化接口、利用接口中的控制函数完成控制操作，最后释放接口。表 7-7 为常用的接口及其功能。

表 7-7 流媒体常用接口

接口	功能
IFilterGraph	滤波器图表接口
IFilterGraph2	增强的 IFilterGraph
IGraphBuilder	最为重要的 COM 接口，用于手动或者自动构造滤波器图表管理器 Filter Graph Manager
ICaptureGraphBuilder2	构造增强型的视频捕获滤波器图表管理器 Filter Graph Manager
IMediaControl	用来控制流媒体，例如流的启动、停止和暂停等，播放控制接口
IMediaEvent	播放事件接口，该接口在 Filter Graph 发生一些事件时用来创建事件的标志信息并传送给应用程序
IMediaEventEx	播放事件接口扩展
IMediaPosition	播放的位置和速度控制接口（控制播放位置只能为设置时间控制方式）
IMediaSeeking	另一个播放的位置和播放速度控制接口，在位置选择方面功能较强。设置播放格式，多种控制播放方式。常用的有：TIME_FORMAT_MEDIA_TIME 单位 100 纳秒和 TIME_FORMAT_FRAME 按帧播放
IBasicAudio	声音控制接口
IBasicVideo	图像控制接口（比特率，图像宽、高等信息）
IBaseFilter	DirectShow 中的基本滤波器接口
IVideoWindow	显示窗口控制接口（有关播放窗口的一切控制，包括标题显示、窗口位置控制等）
ISampleGrabber	捕获图像接口（可用于抓图控制）
IVideoFrameStep	控制单帧播放的接口
ICreateDevEnum	创建设备枚举接口（视频采集用）
IEnumMoniker	枚举监视器接口（视频采集用）
IMoniker	监视接口（视频采集用）
IPropertyBag	属性页接口

7.4 DirectShow SDK常用开发案例

前面介绍了 DirectShow 的工作原理和工作过程，下面介绍在使用 DirectShow SDK 开发应用程序时，经常使用的技术及注意事项。

7.4.1 通用开发流程

使用 DirectShow 开发应用程序，一般可以分为以下 3 个过程。

- 创建滤波器图表管理器。
- 创建滤波器链路。
- 使用滤波器的方法完成流媒体的控制。

针对不同的媒体应用程序，开发流程略有区别，但主要的工作流程大同小异。另外，DirectShow SDK 的$InstallDir\DXSDK\Samples\C++\DirectShow\目录下有各种流媒体的开发应用案例，在开发自己的应用程序时，这些例程都是很好的参考。

7.4.2 系统初始化

开发流媒体应用程序前，系统的初始化是必须的。即在用户源文件中头文件 dshow.h、streams.h 等的包含，库文件 strmiids.lib、strmbasd.lib、uuid.lib 等的引入。加入方法及其作用，在前面我们已经详细地介绍过。

DirectShow 使用了流行的 COM 技术，所以调用 COM 函数前需调用 CoInitialize 函数进行 COM 库的初始化，而结束 COM 库时（退出应用程序）调用 CoUninitialize 函数进行资源释放。下面的例程说明了如何使用 CoUninitialize 函数释放 COM 库。

```
void CMediaPlayerDlg::OnBnClickedButtonExit()
{
//TODO: 在此添加控件通知处理程序代码
OnBnClickedButtonStop();                  //停止播放媒体文件
CoUninitialize();                         //释放COM库
CDialog::OnOK();
}
```

7.4.3 媒体播放方法

流媒体应用中流媒体的播放是应用最多、最广泛的任务。这里只介绍使用 DirectShow SDK 进行流媒体开发中的关键技术部分，具体使用过程请参考后续章节的案例。打开文件流媒体文件，播放并显示图像是基本的功能，一般的方法如下。

```
//常用接口定义
IGraphBuilder    *pGraph;           //滤波器链表创建器
IMediaControl    *pMC;              //媒体控制，如run、pause、stop等
IMediaEventEx    *pME;              //媒体消息接口，媒体的运行状态
IBasicVideo      *pBVideo;          //视频接口
IBasicAudio      *pBAudio;          //音接口
```

```
IVideoWindow     *pVW;                    //视频窗口接口
IMediaSeeking    *pMS;                    //媒体定位接口
//创建Filter Graph Manager组件
CoCreateInstance(CLSID_FilterGraph,       //要创建的Filter的Class ID
            NULL,                         //表示Filter不被聚合
            CLSCTX_INPROC_SERVER,         //创建进程内组件对象
            IID_IGraphBuilder,            //创建Filter成功后获得的接口ID
            (void **)&pGraph)));          //生成的接口对象的指针
if (pGraph)
{
    HRESULT hr = NOERROR;
    //获取各种接口
    hr |= pGraph->QueryInterface(IID_IMediaControl,   (void **)&pMC);
    hr |= pGraph->QueryInterface(IID_IMediaEventEx,   (void **)&pME);
    hr |= pGraph->QueryInterface(IID_IBasicVideo,         (void **)&pBVideo);
    hr |= pGraph->QueryInterface(IID_IBasicAudio,     (void **)&pBAudio);
    hr |= pGraph->QueryInterface(IID_IVideoWindow,    (void **)&pVW);
    hr |= pGraph->QueryInterface(IID_IMediaSeeking,   (void **)&pMS);
}
//…输入播放的流媒体文件路径名：szFilePath
//创建完整的Filter链路
pGraph->RenderFile(szFilePath, NULL));
//设置显示的窗口属性
SetDisplayWindow();
//设置窗口消息通知模式
pME->SetNotifyWindow((OAHWND)inWindow, WM_GRAPHNOTIFY, 0);
//…
//流媒体各种控制方法
pMC->Run();//运行
pMC->Pause();//暂停
pMC->Stop();//停止
//…
//释放资源
pMS->Release();
pMC->Release();
pME->Release();
pBVideo->Release();
pBAudio->Release();
if (pVW)
{
    pVW->put_Visible(OAFALSE);
    pVW->put_MessageDrain((OAHWND)NULL);
    pVW->put_Owner(OAHWND(0));
    pVW->Release();
    pVW = NULL;
}
```

```
pGraph->Release();
//…
```

流媒体在播放中，如何清楚掌握流媒体播放的完毕、暂停状态，应用程序需要得到这些通知。滤波器状态发生变化，视频窗口发生变化等都会产生对应的事件或消息。

7.4.4 消息处理方法

这里也是给出信息处理的关键技术部分，具体使用过程请参考后续章节的案例。常用的事件处理接口为 ImediaEventX，它提供了多种方法以处理 DirectShow 向应用程序发送的消息。事件处理过程和方法如下。

```
//滤波器链表通知给专门的窗口
#define WM_GRAPHNOTIFY  (WM_USER+20)
//头文件中声明
afx_msg LRESULT OnGraphNotify(WPARAM inWParam, LPARAM inLParam);
//实现文件中添加消息映射宏
ON_MESSAGE(WM_GRAPHNOTIFY, OnGraphNotify)
//消息处理函数
LRESULT CPlayerDlg::OnGraphNotify(WPARAM inWParam, LPARAM inLParam)
{
if (pME)                        //媒体事件接口不为空
{
    LONG eventCode = 0;  //事件码
    LONG eParam1 = 0;    //事件的第一个参数
    LONG eParam2 = 0;    //事件的第二个参数
    //读取事件，如果成功，则进行事件的相关处理
    while (SUCCEEDED(pME->GetEvent(&eventCode, &eParam1, &eParam2, 0)))
    {
        pME->FreeEventParams(eventCode, eParam1, eParam2);
        switch (eventCode)  //根据事件码的内容分别处理
        {
        case EC_COMPLETE:
            //播放完毕处理代码
            break;
        case EC_USERABORT:
            //用户终止处理代码
            break;
        case EC_ERRORABORT:
            //出错终止处理代码
            break;
        default:
            break;
        }
    }
}
```

```
    return 0;
}
```

在媒体事件不为空的前提下，首先查询事件码和事件的两个参数。如果查询成功，则释放事件参数，然后处理各个事件码：播放结束（EC_COMPLETE）、用户终止（EC_USERABORT）、出错终止（EC_ERRORABORT）等。如用户可以在媒体播放结束时，重新从头开始播放。

7.4.5 视频采集技术

DirectShow SDK 为音/视频的采集捕获提供了一套简单而功能强大的机制。捕获前，首先需要清楚系统已经安装的采集设备清单，并能够控制其数据格式等。我们希望在捕获保存的同时还能够预览视频。采集设备的格式一般都希望能够配置和修改。以下就以实际的案例程序分析这些技术。

1. 视频采集概览

能够捕捉音/视频的链表 Graph 称为采集 Graph，采集 Graph 比一般的文件回放 Graph 要复杂许多，DirectShow 为我们提供了一个 Capture Graph Builder COM 组件使得采集 Graph 的控制更加简单。Capture Graph Builder 提供了一个 ICaptureGraphBuilder2 接口，这个接口提供了方法用来构建和控制采集 Graph。在创建 Capture Graph Builder COM 对象和 Filter Graph Manger COM 对象后，用后者作为参数调用 ICaptureGraphBuilder2::SetFiltergraph 来初始化 Capture Graph Builder，从而构建了一个采集 Graph。采集 Graph 建立的程序实现如下。

```
//Function:初始化ICaputreGraphBuilder
//利用ICaptureGraphBuilder2 Object帮助我们管理Graph Filter
HRESULT InitCaptureGraphBuilder(
        IGraphBuilder **ppGraph,            //接收返回的指针：链表管理器
        ICaptureGraphBuilder2 **ppBuild     //接收返回的指针：捕获链表管理器
)
{
    if (!ppGraph || !ppBuild)
      return E_POINTER;
    ICaptureGraphBuilder2 *pBuild = NULL;
    //Step 1: 建立Capture Graph Builder.
HRESULT hr = CoCreateInstance(CLSID_CaptureGraphBuilder2, NULL,
                              CLSCTX_INPROC_SERVER,
                              IID_ICaptureGraphBuilder2, (void**)&pBuild);
IGraphBuilder *pGraph = NULL;
    if (SUCCEEDED(hr)){
    //Step 2: 建立Filter Graph Manager.
      hr = CoCreateInstance(CLSID_FilterGraph, NULL,
                      CLSCTX_INPROC_SERVER,
                  IID_IGraphBuilder, (void**)&pGraph);
      if (SUCCEEDED(hr)){
```

```
        //Step 3: 设定pBuild要管理的目标为 Filter Graph Manager
                pBuild->SetFiltergraph(pGraph);
        //Step 4: 将指针回传
                *ppBuild = pBuild;
                *ppGraph = pGraph;      //主调函数需要释放这两个接口
                return S_OK;
            }
            else {
                pBuild->Release();
            }
        }
        return hr;              //创建失败
}
```

上述代码的主要实现流程如图 7-32 所示。

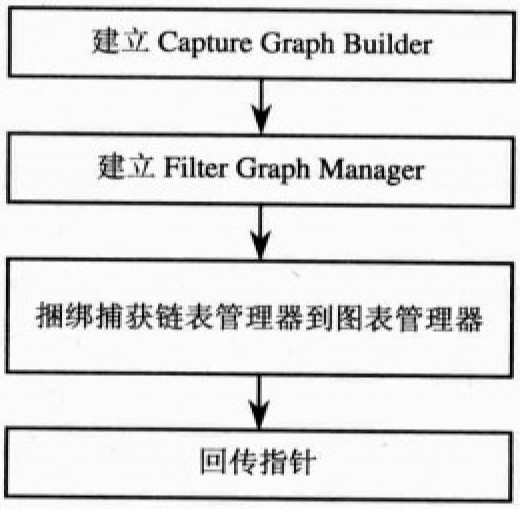


图 7-32　创建采集滤波器链表过程

DirectShow 中所有工作的组件均是 Filter，采集设备在这里称为采集滤波器。Filter 有引脚（Pin），采集滤波器一般有两个或多个引脚，所有引脚的媒体输出类型都相同，如预览引脚（Preview Pin）和捕获引脚（Caputure Pin）。所以再根据媒体类型则不能很好地区别这些引脚了，此时要根据引脚的功能来区别每个引脚，每个引脚都有一个唯一的 ID GUID，称为引脚的种类。对于多数的流媒体应用，ICaptureGraphBuilder2 提供了方法来自动确定引脚的种类。采集滤波器都提供了预览和捕捉的输出引脚，预览引脚用来将视频流在屏幕上显示，捕捉引脚用来将视频流写入文件。预览引脚的种类 GUID 为 PIN_CATEGORY_PREVIEW，捕捉引脚的种类 GUID 为 PIN_CATEGORY_CAPTURE。在实际应用中，如果使用 ICaptureGraphBuilder2 来创建 Capture Graphs，这些 Filters 就会自动被添加到 Graph 中。

2. 采集设备枚举

在开发流媒体采集应用程序前，需要首先确定系统中安装了哪些音/视频采集设备。DirectShow SDK 为选择视频采集设备使用系统设备枚举（System Device Enumerator）。Enumerator 根据采集滤波器的种类返回设备的 Moniker，Moniker 实质是一个 Com 对象接口 IMoniker。一般的采集设备为音频和视频设备，其类标识分别为 CLSID_ AudioInput DeviceCategory 和 CLSID_VideoInputDeviceCategory。

如何枚举视频采集设备，下面的代码演示了基本过程。

```
    ICreateDevEnum *pDevEnum = NULL;       //创建设备枚举接口指针
    IEnumMoniker *pEnum = NULL;            //枚举器监控接口指针
    //创建系统设备枚举器
    HRESULT hr = CoCreateInstance(CLSID_SystemDeviceEnum, NULL,
                                  CLSCTX_INPROC_SERVER,
                                  IID_ICreateDevEnum,
                                  (void**)(&pDevEnum));
    if (SUCCEEDED(hr))
    {//创建一个枚举器，枚举视频设备
    hr = pDevEnum->CreateClassEnumerator(
                            CLSID_VideoInputDeviceCategory,
                            &pEnum, 0);
    }
```

IEnumMoniker 接口 pEnum 返回一个 IMoniker 接口的列表，代表一系列的 Moniker，程序可以显示所有的采集设备，然后让用户选择其一。

一旦选定了某个采集设备，使用 IMoniker::BindToStorage 的方法，返回 IPropertyBag 接口指针，然后调用 IPropertyBag::Read 读取 Moniker 的属性。包含的属性如下。

- FriendlyName：设备的友好名称。
- Description：仅仅适用于 DV 和 D-VHS/MPEG 摄像机，若该属性可用，则此属性更详细地描述了设备的资料。
- DevicePath：该属性是不可读的，并且每个设备都有独一无二的标识。可以用这个属性来区别同一个设备的不同实例。

下面的程序演示了显式遍历设备名称的过程。

```
    HWND hList;                        //列表框句柄
    IMoniker *pMoniker = NULL;
    while (pEnum->Next(1, &pMoniker, NULL) == S_OK)
    {
    IPropertyBag *pPropBag;
    hr = pMoniker->BindToStorage(0, 0, IID_IPropertyBag, (void**)(&pPropBag));
    if (FAILED(hr)) {
        pMoniker->Release();           //如果失败则释放IMoniker接口指针
        continue;                      //跳过该设备，查询下一个
      }
    //查询Description 或者 FriendlyName
    VARIANT varName;
    VariantInit(&varName);
    hr = pPropBag->Read(L"Description", &varName, 0);//以描述型（Description）读取属
性页
    if (FAILED(hr)){                   //若上述操作失败，则以FriendlyName读取属性页
        hr = pPropBag->Read(L"FriendlyName", &varName, 0);
    }
    if (SUCCEEDED(hr))
    {
```

```
        //添加设备名称到列表框
        USES_CONVERSION;                    //宏定义，用于编码转换如W2A,A2W使用的一些局步变量
        //向主窗口发送消息：向hList窗口添加字符串
        (long)SendMessage(hList, LB_ADDSTRING, 0, (LPARAM)OLE2T(varName.bstrVal));
        VariantClear(&varName);             //释放varName资源
    }
    pPropBag->Release();                    //释放属性页IPropertyBag
    pMoniker->Release();                    //释放监控接口IMoniker
    }
```

用户选中某设备后，调用 IMoniker::BindToObject 方法为设备构建滤波器，然后将滤波器加入到 Graph 图中。

```
    IBaseFilter *pCap = NULL;               //基本类型滤波器
    //把基类型滤波器捆绑到设备中，以作为捕获滤波器
    hr = pMoniker->BindToObject(0, 0, IID_IBaseFilter, (void**)&pCap);
    if (SUCCEEDED(hr))
    {
        hr = m_pGraph->AddFilter(pCap, L"Capture Filter"); //把捕获滤波器添加到graph中
    }
```

3. 预览视频数据

在捕获视频时，往往是先浏览视频，根据内容或自己的不同要求，然后再开始捕获、保存视频数据。预览视频数据即把采集到的视频直接写屏幕显示，为了预览视频数据，可以使用下面简单的方法。

```
     &MEDIATYPE_Video, pCap, NULL, NULL);
    ICaptureGraphBuilder2 *pBuild;          //采集图表构建器Capture Graph Builder
      /*…初始化pBuild（没有列出）*/
    IBaseFilter *pCap;                      //视频采集滤波器
      /*…初始化pCap，并添加到Filter graph中（没有列出）*/
    hr = pBuild->RenderStream(&PIN_CATEGORY_PREVIEW,
```

4. 捕获到文件

如果希望保存采集到的视频数据，则可以以多种格式进行保存：原始视频（AVI）、压缩后的视频流（MPG、WMV）等。如图 7-33 所示是一个采集视频保存为 AVI 格式的 Graph 图。

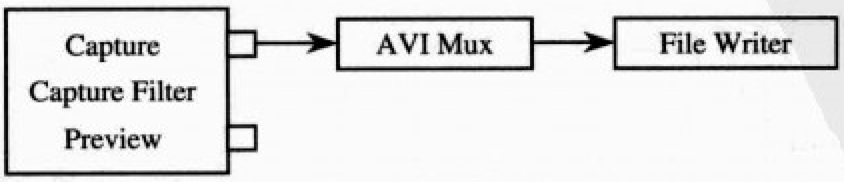


图 7-33　AVI 保存 Graph 图

AVI Mux 滤波器接收从 Capture 引脚来的视频流，然后将其封装为 AVI 流。另外音频流也可以连接到 AVI Mux 滤波器上，这样 Mux 滤波器就将音/视频流合成 AVI 流，File Writer 滤波器将 AVI 流写入到文件中。以下程序构建 AVI 保存 Graph 图。

```
    IBaseFilter *pMux;                      //基类滤波器
```

```
HRESULT hr;                                 //返回值
hr = pBuild->SetOutputFileName(
        &MEDIASUBTYPE_Avi,                  //表明保存为AVI格式的文件
        L"C:\\Example.avi",                 //文件名
        &pMux,                              //返回指向一个Mux指针
        NULL);                              //返回一个文件接收指针(可选)
```

该函数的原型：

```
HRESULT SetOutputFileName(
  const GUID *pType,
  LPCOLESTR lpwstrFile,
  IBaseFilter **ppf,
  IFileSinkFilter **pSink
);
```

其中参数 pType 表明待保存的文件类型，这里是 AVI。参数 lpwstrFile 是文件的名称。对于 AVI 文件，SetOutputFileName 方法会创建一个 AVI Mux 滤波器和一个 File Writer 滤波器，并且将这两个 Filter 添加到 Graph 图中，在此函数中，通过 File Writer 滤波器请求 IFileSinkFilter 接口，然后调用 IFileSinkFilter::SetFileName 方法以设置文件的名称，最后将两个 Filter 连接起来。参数 ppf 返回一个指向 AVI Mux 的指针，同时也通过参数 pSink 返回一个 IFileSinkFilter 指针，如果不需要这个参数，则可以将此参数设置成 NULL。

紧接以上程序，应该调用下面的函数将 Capture Filter 和 AVI Mux 连接起来。

```
hr = pBuild->RenderStream(
        &PIN_CATEGORY_CAPTURE,              //引脚类型为捕获Capture
        &MEDIATYPE_Video,                   //媒体类型为视频
        pCap,                               //捕获滤波器
        NULL,                               //中间滤波器 (可选)
        pMux);                              //Mux 或File Sink 滤波器
  // 释放资源
  pMux->Release();
```

该函数的原型如下。

```
HRESULT RenderStream(
  const GUID *pCategory,                    //引脚类型
  const GUID *pType,                        //媒体类型
  IUnknown *pSource,                        //捕获的源滤波器
  IBaseFilter *pIntermediate,               //中间滤波器（可选）
  IBaseFilter *pSink                        //mux或file sink 滤波器
);
```

上述函数的参数 pSink 就是使用 SetOutputFileName 返回的 pMux 指针。当捕捉音频的时候，媒体类型要设置为 MEDIATYPE_Audio。若要从两个不同的设备捕捉音频和视频，最好将音频设置成主流 Master Stream，这样可以防止两个数据流产生漂移（即不同步），因为 AVI Mux 滤波器为音频同步，并会调整视频的播放速度。为了设置 Master，调用 IConfigAviMux::SetMasterStream 方法，可采用如下代码。

```
IConfigAviMux *pConfigMux = NULL;     //AVI复用配置接口
```

```
//获取复用配置接口
hr = pMux->QueryInterface(IID_IConfigAviMux, (void**)&pConfigMux);
if (SUCCEEDED(hr))
{
    pConfigMux->SetMasterStream(1);    //设置主同步数据流
    pConfigMux->Release();             //设置完毕释放该接口
}
```

SetMasterStream 的参数指的是主同步数据流的索引，这个是由调用 RenderStream 的次序决定的。例如，如果调用 RenderStream 时首先用于视频流，然后是音频，那么视频流是 0，音频流是 1。

如果希望把采集到的视频编码压缩后再写文件，添加编码滤波器过程如下。

```
IBaseFilter *pEncoder;
 /*…创建编码滤波器(没有列出).*/
 //添加编码滤波器到滤波器Graph中
 pGraph->AddFilter(pEncoder, L"Encoder);
 /*如前所述调用SetOutputFileName*/
 //渲染媒体流
 hr = pBuild->RenderStream(&PIN_CATEGORY_CAPTURE,
                    &MEDIATYPE_Video,
                    pCap, pEncoder, pMux);
pEncoder->Release();
```

在预览视频的同时又能够采集捕获到文件，这是很正常的需求。则需要创建一个预览、保存文件的 Graph 图，DirectShow SDK 为我们提供了简单的方法以完成此任务，即两次调用 ICaptureGraphBuilder2::RenderStream 方法。过程如下。

```
 //把预览视频流渲染到显示器
 hr = pBuild->RenderStream(&PIN_CATEGORY_PREVIEW,
                    &MEDIATYPE_Video, pCap, NULL, NULL);
 //把采集视频流渲染到写文件Mux
 hr = pBuild->RenderStream(&PIN_CATEGORY_CAPTURE,
                    &MEDIATYPE_Video, pCap, NULL, pMux);
```

5. 控制捕获链表

滤波器链表管理器可以通过 IMediaControl 接口控制整个 Graph 图的运行、停止和暂停。但是当 Graph 图既有捕获又有预览两个数据流的时候，如果我们想单独地控制其中一个数据流，我们可以通过 ICaptureGraphBuilder2::ControlStream 方法。下面我们来分析如何单独控制捕获和预览数据流。

1）控制捕获视频流

下面的代码演示了捕捉数据流在 Graph 开始运行 1 秒后开始，运行 5 秒后结束。

```
//控制视频捕获流
 REFERENCE_TIME rtStart = 1000 0000, rtStop = 6000 0000;
 const WORD wStartCookie = 1, wStopCookie = 2;    //任意值
 hr = pBuild->ControlStream(
```

```
        &PIN_CATEGORY_CAPTURE,                    //引脚类型
        &MEDIATYPE_Video,                         //媒体类型
        pCap,                                     //捕获Filter
        &rtStart, &rtStop,                        //开始和停止的时间
        wStartCookie, wStopCookie);               //开始和停止时间代码值
    pControl->Run();
```

ControlStream 的函数原型如下。

```
HRESULT ControlStream(
  const GUID *pCategory,
  const GUID *pType,
  IBaseFilter *pFilter,
  REFERENCE_TIME *pstart,
  REFERENCE_TIME *pstop,
  WORD wStartCookie,
  WORD wStopCookie
);
```

第 1 个参数 pCategory 表明需要控制的数据流，一般采用的是引脚种类 GUID。

第 2 个参数 pType 表明了媒体类型。

第 3 个参数 pFilter 指明了捕获的滤波器。如果想要控制 Graph 图中的所有捕获滤波器，则第 2 个和第 3 个参数都要设置成 NULL。

第 4 个参数 pstart 和第 5 个参数 pstop 表明了流开始和结束的时间，这是一个相对于 Graph 开始的时间。只有在调用 IMediaControl::Run 方法后，此函数才起作用。如果 Graph 正在运行，则此设置立即生效。

最后的两个参数 wStartCookie、wStopCookie 用来设置当数据流停止时，开始能够得到的事件通知。对于任何一个运用此方法的数据流，Graph 当流开始的时候，会发送 EC_STREAM_CONTROL_STARTED 通知，在流结束的时候，要发送 EC_STREAM_CONTROL_STOPPED 通知。wStartCookie 和 wStopCookie 是作为第二个事件参数的。

其事件处理过程如下。

```
//获取事件码和事件的两个参数
while (hr = pEvent->GetEvent(&evCode, &param1, &param2, 0), SUCCEEDED(hr))
{
switch (evCode)
{
  case EC_STREAM_CONTROL_STARTED:     //处理事件
  // param2 == wStartCookie
   break;
 case EC_STREAM_CONTROL_STOPPED:      //处理事件
 // param2 == wStopCookie
    break;
 }
pEvent->FreeEventParams(evCode, param1, param2);//释放事件参数
}
```

ControlStream 还定义了一些特定的值来表示开始和停止的时间。若 Start 和 Stop 为 MAXLONGLONG，则从不开始，只有在 Graph 停止的时候才停止；若 Start 和 Stop 为 NULL，则立即开始和停止。

2）控制预览视频流

只要将ControlStream的第一个参数设置为PIN_CATEGORY_PREVIEW就可以控制预览引脚，并且整个函数的使用和控制捕捉流一样，但是唯一的区别是在这里我们没法设置开始和结束时间，因为预览的视频流没有时间戳，因此必须使用 NULL 或者 MAXLONGLONG。

```
//使用NULL开始预览视频流
  pBuild->ControlStream(&PIN_CATEGORY_PREVIEW, &MEDIATYPE_Video, pCap,
  NULL,           //马上开始
  0,              //保留
  wStartCookie, wStopCookie);
    //使用 MAXLONGLONG 停止预览视频流
  pBuild->ControlStream(&PIN_CATEGORY_PREVIEW, &MEDIATYPE_Video, pCap,
  0,              //保留
  MAXLONGLONG,    //马上停止
  wStartCookie, wStopCookie);
```

6. 配置捕获设备

视频采集设备根据驱动模式可分为 VFW 和 WDM 两种，目前应用最广泛的是 WDM 模式。下面分别对这两种模式的采集设备的配置进行介绍。

1）VFW 驱动采集设备配置

如果视频捕捉设备驱动模式是 VFW 方式，则必须支持下面 3 个对话框用来设置。

- 视频源（Video Source）：用来选择视频输入设备并且调整设备的参数配置，如画面亮度和对比度等。
- 视频格式（Video Format）：用来设置视频帧的大小和比特位宽度。
- 视频显示（Video Display）：用来设置视频的显示参数。

为了显示上面的 3 个对话框，需要按下面步骤操作。

STEP 01 停止运行 Graph。

STEP 02 向捕捉 Filter 请求 IAMVfwCaptureDialogs 接口，若成功则表明设备支持 VFW 驱动。

STEP 03 调用 IAMVfwCaptureDialogs::HasDialog 来检查驱动程序是否支持请求的对话框，若支持则返回 S_OK，否则返回 S_FALSE。注意不要用 SUCCEDED 宏。

STEP 04 如果驱动支持该对话框，则调用 IAMVfwCaptureDialogs::ShowDialog 显示该对话框。

STEP 05 重新运行 Graph。

程序实现上述过程如下。

```
pControl->Stop();          //停止运行Graph
```

```
//查询 IAMVfwCaptureDialogs 接口
IAMVfwCaptureDialogs *pVfw = 0;
hr = pCap->QueryInterface(IID_IAMVfwCaptureDialogs, (void**)&pVfw);
if (SUCCEEDED(hr))
{
  //检查是否支持要请求的对话框
  if (S_OK == pVfw->HasDialog(VfwCaptureDialog_Source))
     {
     //显示该对话框
     hr = pVfw->ShowDialog(VfwCaptureDialog_Source, hwndParent);
     }
}
pControl->Run();          //重新运行Graph
```

WDM 模式驱动的采集设备支持的一些属性可以用来调整视频图像的质量，如亮度、对比度和饱和度等。预调整视频的质量，步骤如下。

STEP 01 从捕捉 Filter 上请求 IAMVideoProcAmp 接口。

STEP 02 对于预调整的任何一个属性，调用 IAMVideoProcAmp::GetRange 可以返回该属性赋值的范围、默认值和最小的增量值等信息。IAMVideoProcAmp::Get 返回当前正在使用的值，VideoProcAmpProperty 枚举每个属性定义的标志。

STEP 03 调用 IAMVideoProcAmp::Set 来设置这个属性值。设置属性的时候，无须停止 Graph。

程序实现上述过程如下。

```
HWND hTrackbar; //trackbar句柄
 //初始化 hTrackbar (没有列出)
 //查询 IAMVideoProcAmp接口
 IAMVideoProcAmp *pProcAmp = 0;
 hr = pCap->QueryInterface(IID_IAMVideoProcAmp, (void**)&pProcAmp);
 if (FAILED(hr))
 {
   //设备不支持 IAMVideoProcAmp,禁用hTrackbar
   EnableWindow(hTrackbar, FALSE);
 } else
 {
   long Min, Max, Step, Default, Flags, Val;
   //得到默认值和范围大小等信息
   hr = m_pProcAmp->GetRange(VideoProcAmp_Brightness,
                        &Min, &Max, &Step,
                        &Default, &Flags);
   if (SUCCEEDED(hr))
   {
     //获取当前值
     hr = m_pProcAmp->Get(VideoProcAmp_Brightness, &Val, &Flags);
   }
```

```
    if (SUCCEEDED(hr))
    {
        //设置滑动条的范围和当前位置
        SendMessage(hTrackbar, TBM_SETRANGE, TRUE, MAKELONG(Min, Max));
        SendMessage(hTrackbar, TBM_SETPOS, TRUE, Val);
        EnableWindow(hTrackbar, TRUE);
    }else
    {
       EnableWindow(hTrackbar, FALSE); //不支持该属性
    }
  }
```

首先获得 IAMVideoProcAmp 接口，如果获取失败，则禁用配置滑动条。如果获取成功，则首先读取默认值和大小范围等信息。然后设置配置滑动条的范围和当前位置。

视频采集设备的输出流可以有多种格式，如可支持 16/32-bit RGB、YUYV、I420 等，对于每种格式，还可以调整视频帧的大小。

在 WDM 模式驱动设备上，IAMStreamConfig 接口用来报告设备输出视频的格式，VFW 设备可以采用对话框的方式来设置，参见前面的内容。

捕捉滤波器的捕捉引脚和预览引脚都支持 IAMStreamConfig 接口，可以通过 ICaptureGraphBuilder2::FindInterface 获得 IAMStreamConfig 接口。代码如下。

```
IAMStreamConfig *pConfig = NULL;
hr = pBuild->FindInterface(
            &PIN_CATEGORY_PREVIEW,       //预览引脚
            0,                           //任何媒体类型
            pCap,                        //指向捕获滤波器
            IID_IAMStreamConfig, (void**)&pConfig);
```

FindInterface 函数根据传入的不同参数查询所支持的功能，可以查询预览或捕获引脚，媒体类型可以是音频或视频。

2）WDM 驱动采集设备配置

目前多数的采集设备如 USB 摄像头、主流视频采集卡等都是 WDM 驱动模式。应用 DirectShow SDK 开发的流媒体程序，如何配置此类设备的视频参数，下面以具体的案例来分析配置过程。

由于在 DirectShow 中，采集设备也视为一个滤波器，即 Capture Source Filter。所以采集设备的属性配置就演变成引脚属性配置和 Capture Filter 属性配置了。引脚属性配置包括视频的颜色空间、输出大小和帧率等，Capture Filter 属性配置即图像的参数（亮度、对比度等）控制。

① 引脚属性配置

配置滤波器的引脚属性，首先寻找 Graph 图中的 IAMStreamConfig 接口，然后再由该接口查询 ISpecifyPropertyPages 接口，从而获得多个引脚的多个属性页。如下代码实现上述过程。

```
void CCaptureVideo::ConfigCameraPin(HWND hwndParent)
{
 HRESULT hr;                           //返回值
```

```
  IAMStreamConfig *pSC;                  //流配置接口
  ISpecifyPropertyPages *pSpec;          //属性页
//只有停止后，才能进行引脚属性的设置
m_pMC->Stop();
//首先查询捕获CAPTURE、视频Video接口。
hr = m_pCapture->FindInterface(&PIN_CATEGORY_CAPTURE,
                    &MEDIATYPE_Video, m_pBF,
                    IID_IAMStreamConfig, (void **)&pSC);
   CAUUID cauuid; //所有属性页结构体
//然后获取属性页
hr = pSC->QueryInterface(IID_ISpecifyPropertyPages, (void **)&pSpec);
if(hr == S_OK)
{
    hr = pSpec->GetPages(&cauuid);    //获取所有属性页
    //显示属性页
    hr = OleCreatePropertyFrame(hwndParent,//tt.m_hWnd, 30, 30, NULL, 1,
        (IUnknown **)&pSC, cauuid.cElems, (GUID *)cauuid.pElems, 0, 0, NULL);
    CoTaskMemFree(cauuid.pElems);
    pSpec->Release();
    pSC->Release();
}
m_pMC->Run();
}
```

上述代码首先以视频捕获方式获取流配置接口，在该接口下查询属性页接口，最后获取所有属性页，然后显示获取的属性页，最后释放资源。上述程序的运行结果如图 7-34 所示。

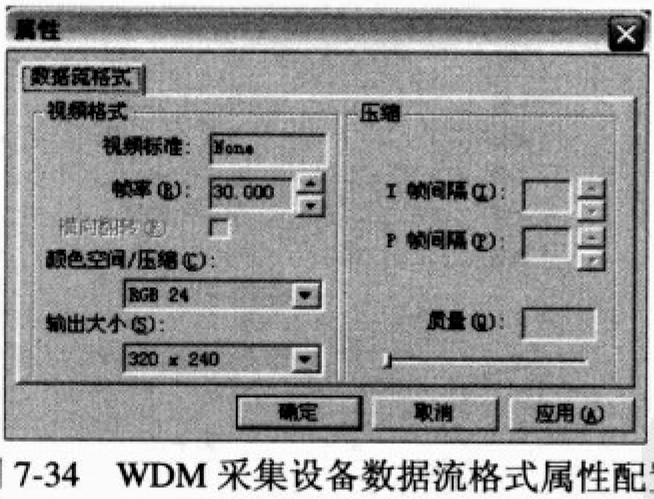


图 7-34　WDM 采集设备数据流格式属性配置

② Filer 属性配置

类似于引脚属性的配置，首先查询 ISpecifyPropertyPages 接口，然后获取滤波器的名字，最后显示该页。代码实现过程如下。

```
void CCaptureVideo::ConfigCameraFilter(HWND hwndParent)
{
HRESULT hr=0;                           //函数返回值
ISpecifyPropertyPages *pProp;           //属性页接口
//在视频捕获滤波器下查询属性页接口
hr = m_pBF->QueryInterface(IID_ISpecifyPropertyPages, (void **)&pProp);
```

```
if (SUCCEEDED(hr))
{
    //获取Filter的名字和 IUnknown接口
    FILTER_INFO FilterInfo;
    hr = m_pBF->QueryFilterInfo(&FilterInfo);
    IUnknown *pFilterUnk;
    m_pBF->QueryInterface(IID_IUnknown, (void **)&pFilterUnk);
    //显示该页
    CAUUID caGUID;
    pProp->GetPages(&caGUID);
    OleCreatePropertyFrame(
          hwndParent, 0, 0, FilterInfo.achName, 1, &pFilterUnk, caGUID.cElems,
          caGUID.pElems, 0,0, NULL);
    //释放资源
    CoTaskMemFree(caGUID.pElems);
    pFilterUnk->Release();
    FilterInfo.pGraph->Release();
    pProp->Release();
}
}
```

运行结果如图 7-35 所示。捕获滤波器属性是对画面的参数进行配置和修改，包括图像参数、白平衡和模式控制。这些参数一旦更改，预览的视频图像马上起作用，当调整到合适的情况下，单击“确定”按钮，关闭该对话框。

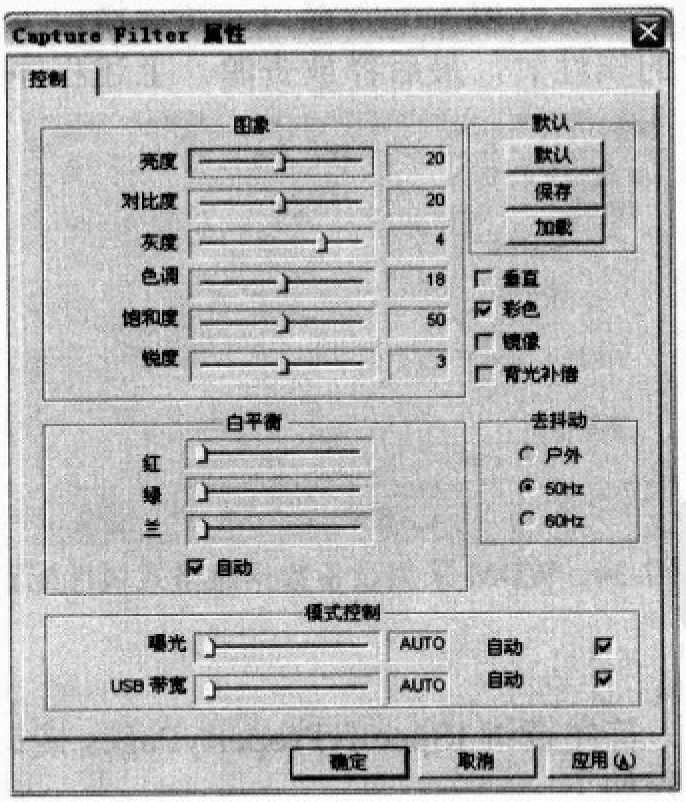


图 7-35　WDM 采集设备 Capture Filter 属性配置

7.4.6　音频采集案例

DirectShow SDK 不但能够采集、播放视频，还能够对音频进行采集或回放。音频文件的

回放同视频的回放非常相似，前面已经详细介绍。这里只对音频的采集、存储进行案例设计和剖析，体会 DirectShow SDK 的流媒体开发的强大功能。

1. Filter 图表设计

DirectShow 为我们提供的 GraphEdit.exe 工具能够快速构建一个由各种滤波器构成的链表，完成流媒体应用。这里再次使用该工具，构建音频采集、存储链表。构建大体过程为插入必需的滤波器，连接滤波器构成链表，最后运行链表。

STEP 01 启动 GraphEdit 程序，路径为 D:\DXSDK\Bin\DXUtils\，启动结果如图 7-36 所示。用户可以发现该目录下是微软预生成的各种有用的可执行程序，这些可执行程序可以帮助用户调试、构建流媒体程序。

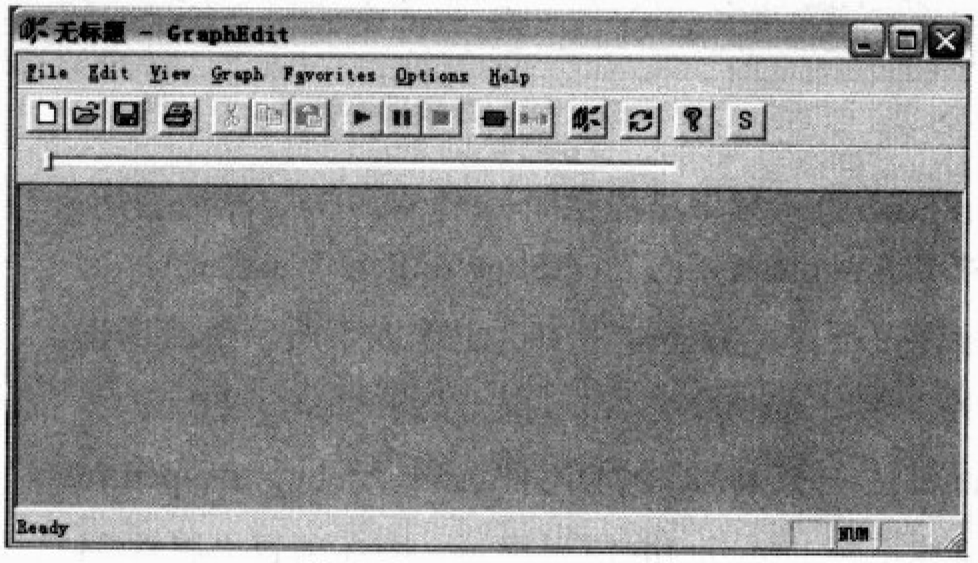


图 7-36　GraphEdit 工具

STEP 02 单击"■"按钮打开插入滤波器对话框，如图 7-37 所示。选中"Audio Capture Source"选项，单击前面的加号，则该选项下列出了本系统已经安装的音频采集设备。

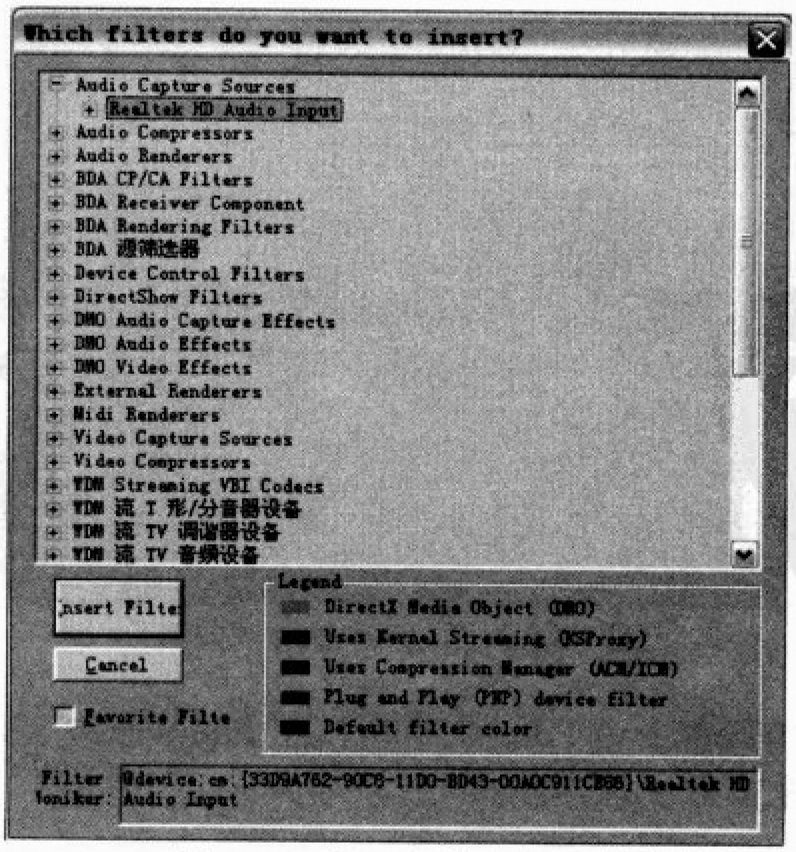


图 7-37　插入滤波器对话框

STEP 03 选择一个音频采集设备，然后单击“Insert Filters”按钮，采集滤波器就添加到视图中，如图 7-38 所示。笔者操作系统已经安装了“Realtek HD Audio Input”音频设备。

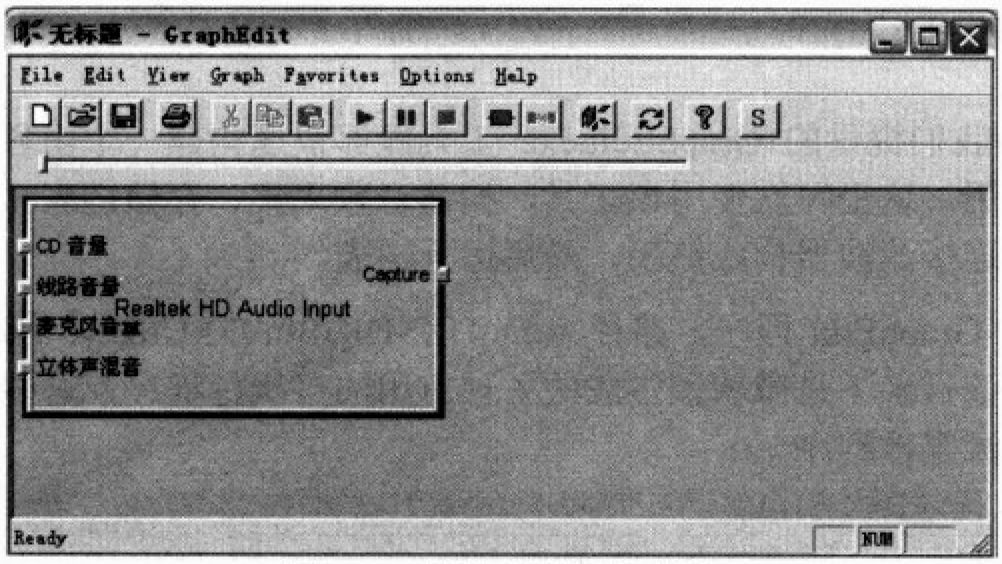


图 7-38　插入音频采集设备

STEP 04 DirectShow 为音频采集提供了一个 WavDest 滤波器用来写 Wav 格式的音频文件，路径为 D:\DXSDK\Samples\C++\DirectShow\Filters\WavDest\。

先用 VC2005 编译该项目，编译过程中会出现以下所示链接错误。

```
fatal error LNK1117: 选项"debug:mapped,full"中的语法错误。
```

我们只要将项目属性/链接器/命令行/附件选项“debug:mapped,full /”删掉即可。

继续编译，出现找不到 largeint.lib 错误提示。则在项目“属性/链接器/输入/附加依赖项”中，删除该库的连接。成功编译后，使用 regsvr32.exe 注册 WavDest.ax 滤波器。

在关闭“插入滤波器对话框”后，重新打开，单击“DirectShow Filters”按钮，选中“Wave Dest”滤波器，如图 7-39 所示。

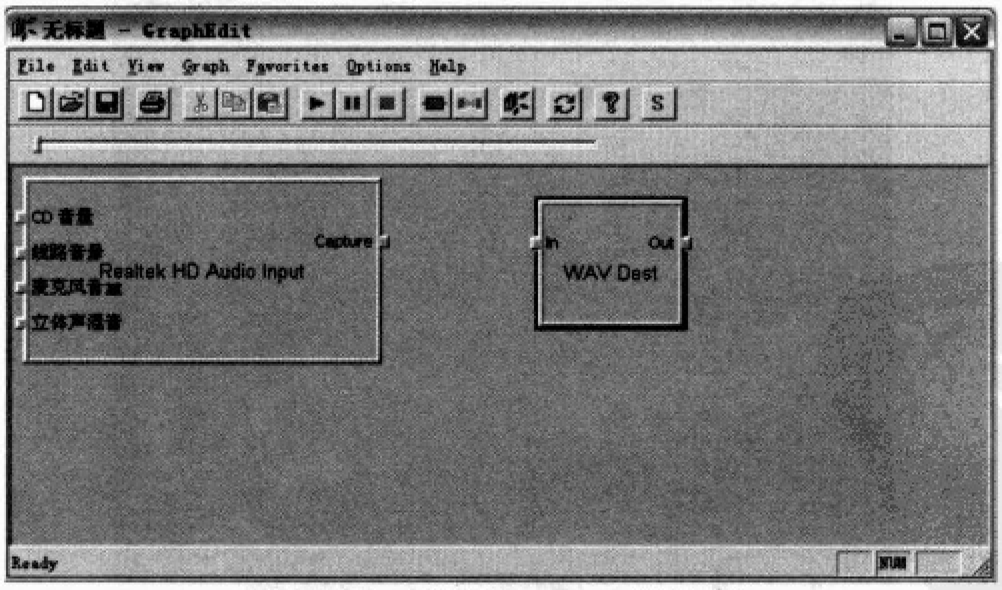


图 7-39　插入 Wav Dest 滤波器

STEP 05 最后添加“DirectShow Filters”节点下的“File Writer”滤波器，单击“Insert Filters”按钮，弹出对话框，如图 7-40 所示。

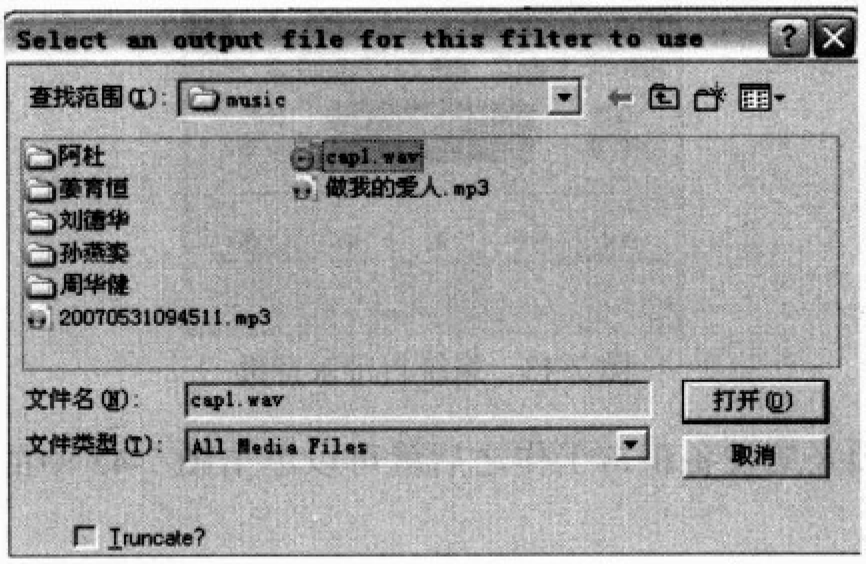


图 7-40　保存文件对话框

STEP 06 插入写文件滤波器后，完成的滤波器图表如图 7-41 所示。

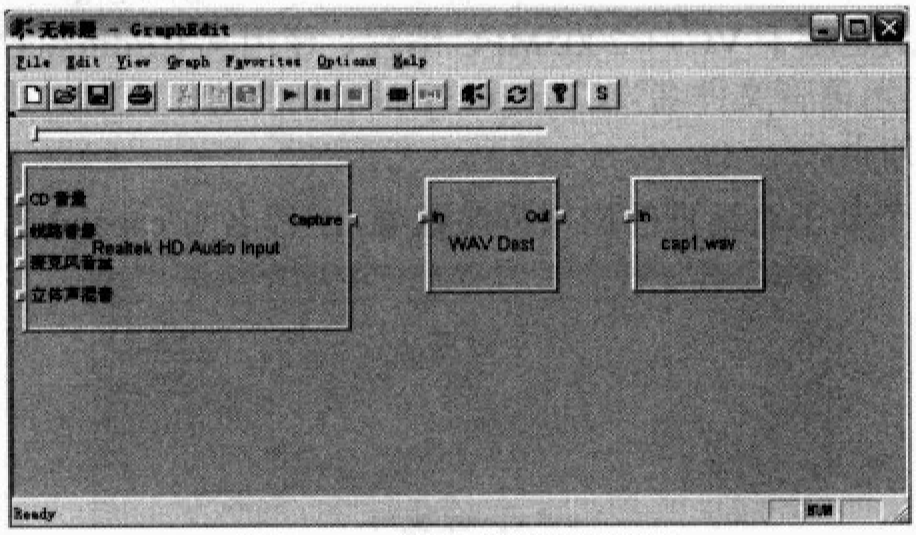


图 7-41　音频采集滤波器图

STEP 07 依次把这 3 个滤波器的引脚根据顺序，从左向右连接起来。使用鼠标把“Caputer”接口与“WAV Dest”的 In 接口连接，“WAV Dest”的 Out 接口与“Filter Writer”的 In 接口连接，连接后的图表如图 7-42 所示。

STEP 08 至此音频采集的滤波器链表已经建立完毕，但在采集前需要确定音频的采集设备是否正常工作。简单的方法是使用操作系统自带的“录音机”，录制、回放音频检查麦克风等采集设备的工作状态，具体图形如图 7-43 所示。

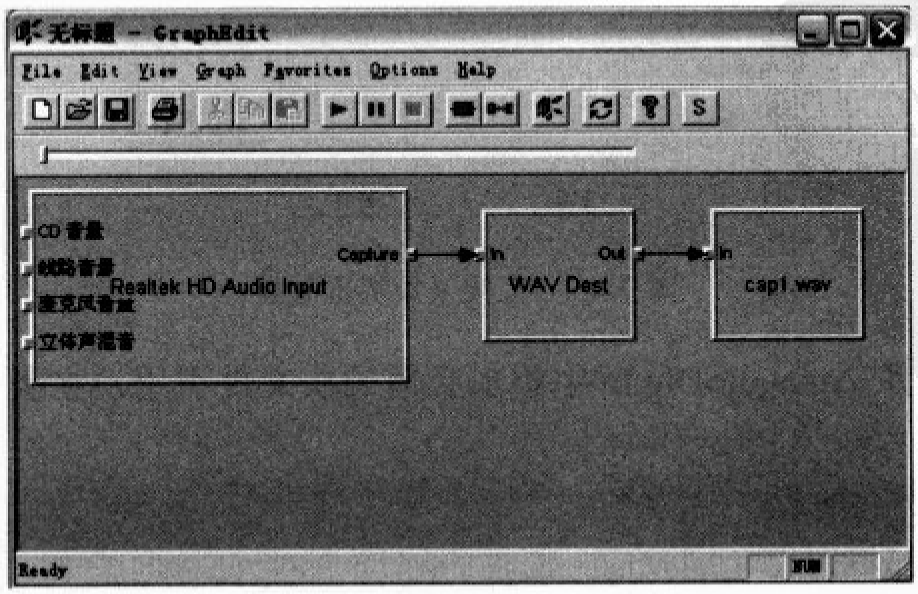


图 7-42　音频采集滤波器链表

图 7-43　系统自带录音机

STEP 09 确认音频采集设备正常工作之后就可以单击图 7-43 中的 ▶| 按钮，开始采集音频并保存文件。

2. 音频案例编程

有了滤波器链表设计流程，就可以根据此过程，使用 DirectShow SDK 编程实现音频的采集存储任务。本案例是基于 VC++ 2005MFC 编程，使用其 CEdit 和 CButton 控件设计一个可视化界面，CEdit 显示语音文件名，CButton 控制采集的动作。案例完成对音频的采集并存储到文件，具体步骤如下。

STEP 01 创建一个基于对话框的应用程序框架，工程名为 capaudio，画面如图 7-44 所示。

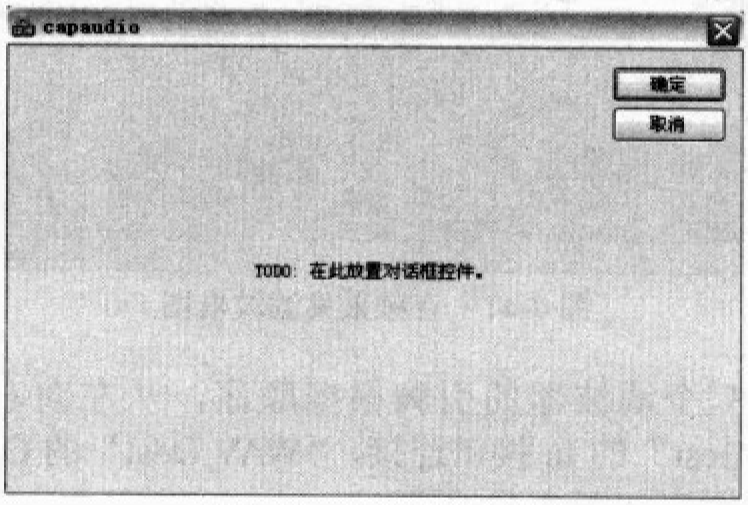


图 7-44　简单应用程序

STEP 02 在 VC 2005 的"资源视图"中对主界面进行编辑，添加控件 Edit Control、Button，设计的画面如图 7-45 所示。

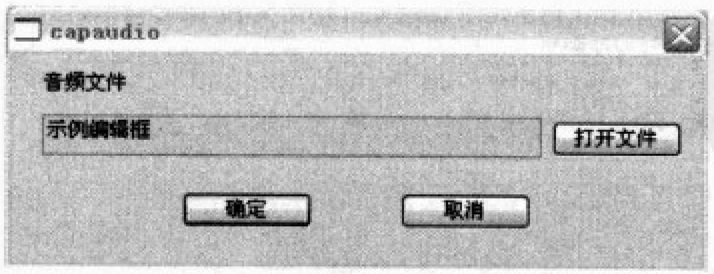


图 7-45　音频采集界面设计

STEP 03 在头文件 capaudioDlg.h 中添加必需的头文件引用，在类中定义必需的指针成员变量。

```
//capaudioDlg.h : 头文件
//添加DirectShow头文件
#include <dshow.h>
```

```
#pragma once
//CcapaudioDlg 对话框
class CcapaudioDlg : public CDialog
{
//构造
public:
CcapaudioDlg(CWnd* pParent = NULL);        //标准构造函数
IGraphBuilder *pGraph;                     //滤波器链表管理器接口
ICaptureGraphBuilder2 * pBuilder;          //捕获链表管理器接口
IMediaControl *pMC;                        //媒体控制接口
IMoniker *pMoniker;                        //设备监控接口
IBaseFilter *pSrc,*pWaveDest,*pWriter;     //基类滤波器接口
IFileSinkFilter2 *pSink;                   //写文件接口
    IPin * FindPin(IBaseFilter *pFilter, PIN_DIRECTION dir);
//对话框数据
enum { IDD = IDD_CAPAUDIO_DIALOG };
protected:
virtual void DoDataExchange(CDataExchange* pDX);        //DDX/DDV 支持
//实现
protected:
HICON m_hIcon;
//生成的消息映射函数
virtual BOOL OnInitDialog();                            //初始化对话框，是虚拟函数
afx_msg void OnSysCommand(UINT nID, LPARAM lParam);     //系统命令处理函数
afx_msg void OnPaint();                                 //绘图函数
afx_msg HCURSOR OnQueryDragIcon();                      //绘制图标函数
DECLARE_MESSAGE_MAP()                                   //以下为消息映射
public:
afx_msg void OnBnClickedRecord();
public:
afx_msg void OnBnClickedCancel();
public:
afx_msg void OnBnClickedStop();
};
```

上述代码包括自己手工编写的和 VC++ 2005 自动生成的代码。例如消息映射函数是 VC 自动添加的。一般前缀有 afx_msg 就表示该函数是消息映射函数。

STEP 04 对编辑控件 Edit Control 捆绑变量，以便于程序中对其进行显示、编辑等操作。首先修改其 ID 为 IDC_EDIT_PATH，选中并用鼠标右键单击控件，如图 7-46 所示。

选择“添加变量”命令，为控件添加变量，如图 7-47 所示，这样的目的是在程序中对 CString 类型变量作修改，直接和编辑控件捆绑，添加成员变量向导如图 7-48 所示。

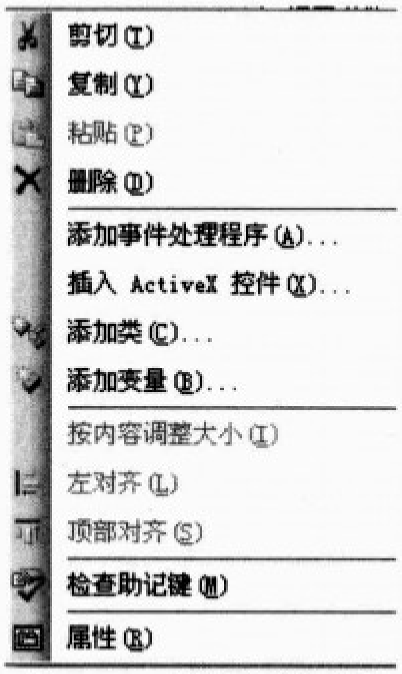


图 7-46　控件鼠标右键

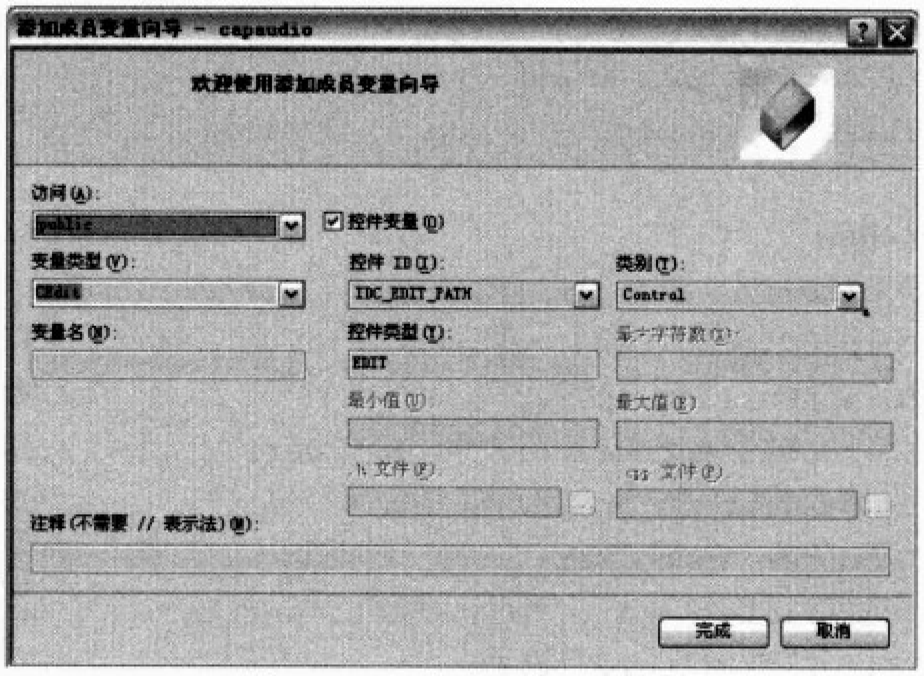


图 7-47　添加成员变量向导

图 7-48　添加成员变量向导

STEP 05 双击“打开文件”按钮，添加按钮单击事件响应处理函数，实现音频文件名的设置。

```
    void CcapaudioDlg::OnBnClickedButtonFileName()
    {
    //TODO: 在此添加控件通知处理程序代码
    CString strFilter = _T("Wav File (*.wav) | *.wav|");    //文件类型过滤表
    strFilter += "All File (*.*) | *.*|";
    //声明对话框对象，参数为只读、路径必须存在等。
    CFileDialog dlg(FALSE, NULL, NULL, OFN_PATHMUSTEXIST|OFN_HIDEREADONLY, strFilter,
this);
    if (dlg.DoModal()==IDOK)
    {
        CString strName = dlg.GetPathName();        //获取选中的文件路径名
        strName += ".wav";                          //为文件名加后缀wav
```

```
    this->m_PathName = strName;                    //记录文件名
    UpdateData(FALSE);                             //把变量的值更新显示到控件。
  }
}
```

STEP 06 在上节中，已经注册了 WavDest.ax 滤波器。为了在源程序中使用 WavDest 滤波器，需要定义该滤波器。在 GraphEdit 中查看该滤波器的 CLSID 值，在 CcapaudioDlg.cpp 中定义：

```
static const GUID CLSID_WavDest = {0x3C78B8E2,0x6C4D,0x11D1,{0xAD,0xE2,0x0,
0x0,0xF8,0x75,0x4B,0x99}};
```

STEP 07 Filter 是 COM 组件，所以必须对 COM 初始化。在 BOOL CcapaudioApp::InitInstance()函数中添加：

```
CoInitialize(NULL);            //初始化COM库，参数为空。
```

STEP 08 记录按钮消息响应函数 OnBnClickedRecord 实现如下。

```
void CcapaudioDlg::OnBnClickedRecord()
{   //TODO: 在此添加控件通知处理程序代码
CString strName;                  //记录的文件名
GetDlgItem(IDC_EDIT_PATH)->GetWindowText(strName);    //获取CEdit中的内容
if(strName.IsEmpty())             //如果文件名为空
{
    MessageBox(_T("请先设置音频文件名! "));                   //提示文件名为空
    return;                                               //返回，请重新选择输入
}
GetDlgItem(ID_STOP)->EnableWindow(true);                  //使能停止按钮
pGraph = NULL;                    //滤波器链表接口
pMC = NULL;                       //媒体控制接口
pBuilder = NULL;                  //捕获链表管理器接口
//获取捕获链表管理器接口
CoCreateInstance(CLSID_CaptureGraphBuilder2,NULL,
                        CLSCTX_INPROC_SERVER,
                        IID_ICaptureGraphBuilder2,
                        (void**)&pBuilder);
//获取滤波器链表接口
CoCreateInstance(CLSID_FilterGraph, NULL,
                CLSCTX_INPROC_SERVER,
               IID_IGraphBuilder,
                (void **)&pGraph);
//把上述接口捆绑
pBuilder->SetFiltergraph(pGraph);
//查询并获取媒体控制接口
pGraph->QueryInterface(IID_IMediaControl,(void**)&pMC);
ICreateDevEnum *pDevEnum = NULL; //创建设备枚举接口
//获取设备枚举接口
CoCreateInstance(CLSID_SystemDeviceEnum, NULL,
```

```
                    CLSCTX_INPROC,
                    IID_ICreateDevEnum,
                    (void **)&pDevEnum);
IEnumMoniker *pClassEnum = NULL;
//创建枚举器监控接口
pDevEnum->CreateClassEnumerator(CLSID_AudioInputDeviceCategory, &pClassEnum, 0);
ULONG cFetched;
if (pClassEnum->Next(1, &pMoniker, &cFetched) == S_OK)    //查询设备
{
    //捆绑到基类滤波器，以作为捕获滤波器
    pMoniker->BindToObject(0, 0, IID_IBaseFilter, (void**)&pSrc);
    pMoniker->Release();
}
pClassEnum->Release();
//查询WavDest类以写Wav格式文件
CoCreateInstance(CLSID_WavDest, NULL, CLSCTX_ALL,
                    IID_IBaseFilter, (void **)&pWaveDest);
//查询写文件FileWriter接口
CoCreateInstance(CLSID_FileWriter, NULL, CLSCTX_ALL,
                    IID_IBaseFilter, (void **)&pWriter);
pGraph->AddFilter(pSrc,L"Wav");                        //添加捕获滤波器，名为Wav
pGraph->AddFilter(pWaveDest,L"WavDest");               //添加WavDest滤波器，名为WavDest
pGraph->AddFilter(pWriter,L"FileWriter");              //添加写文件滤波器，名为FileWriter
pWriter->QueryInterface(IID_IFileSinkFilter2,(void**)&pSink);
pSink->SetFileName(strName.AllocSysString(),NULL);//设置捕获的文件名
IPin* pOutpin = FindPin(pSrc,PINDIR_OUTPUT); //查询捕获的输出引脚
IPin* pInpin,*pOut;
pOut= FindPin(pWaveDest,PINDIR_OUTPUT);                //查询WavDest的输出引脚
AM_MEDIA_TYPE type;                                    //媒体类型
type.majortype = MEDIATYPE_Stream;                     //主类型为流媒体
type.subtype =MEDIASUBTYPE_WAVE;                       //子类型为Wave
type.formattype = FORMAT_None;                         //格式为空
type.bFixedSizeSamples = FALSE;                        //非固定样本大小
type.bTemporalCompression = FALSE;                     //时域不压缩
type.pUnk = NULL;
   //查找滤波器引脚
pInpin = FindPin(pWaveDest,PINDIR_INPUT);
IPin* pInpin2= FindPin(pWriter,PINDIR_INPUT);
   //连接滤波器的引脚
//捕获的输出引脚连接到WavDest的输入引脚
pGraph->ConnectDirect(pOutpin,pInpin,NULL);
WavDest的输出引脚连接到写文件的输入引脚
pGraph->ConnectDirect(pOut,pInpin2,NULL);
pMC->Run();                                            //媒体开始运行
GetDlgItem(ID_RECORD)->EnableWindow(false);  //使能记录按钮
```

```
GetDlgItem(ID_STOP)->EnableWindow(true);      //使能停止按钮
}
```

上述的采集、保存音频的过程非常类似于视频的采集和保存，只是中间用到了DirectShow提供的一个中间滤波器WavDest接口，它主要用来把语音数据编码成Wave格式。建立采集链表的过程中，用户需特别注意查询滤波器的引脚函数FindPin()，它是根据输入的引脚方向参数来查询输入的滤波器引脚的。

STEP 09 函数FindPin是查询滤波器指定的引脚是否存在。

```
IPin* CcapaudioDlg::FindPin(IBaseFilter *pFilter, PIN_DIRECTION dir)
{
IEnumPins* pEnumPins;                        //引脚枚举接口
IPin* pOutpin;                               //引脚接口
PIN_DIRECTION pDir;                          //引脚方向
pFilter->EnumPins(&pEnumPins);               //枚举滤波器的所有引脚
//根据索引序号查询引脚
while (pEnumPins->Next(1,&pOutpin,NULL)==S_OK)
{
    pOutpin->QueryDirection(&pDir);  //查询引脚的方向
    if (pDir==dir)
    {
        return pOutpin;                      //查询到指定引脚的方向
    }
}
return 0;
}
```

上述函数首先枚举传入的滤波器的所有引脚，然后返回指定索引的引脚，如果成功则再查询该引脚的方向，如果与传入的引脚方向相同则返回，否则返回失败0值。

STEP 10 录制完成，OnBnClickedStop函数停止录音。

```
void CcapaudioDlg::OnBnClickedStop()
{
//TODO: 在此添加控件通知处理程序代码
pMC->Stop();                 //停止媒体
pMC->Release();              //释放媒体控制接口
pGraph->Release();           //释放滤波器链表接口
pWaveDest->Release();        //释放WavDest接口
pWriter->Release();          //释放写文件接口
MessageBox(_T("录制完成"));
GetDlgItem(IDC_EDIT_PATH)->SetWindowText(_T(""));      //把CEdit内容清空
GetDlgItem(ID_STOP)->EnableWindow(false);              //禁用停止按钮
}
```

当停止录音时，需要把所有申请的指针、控件释放，以便于下次继续控制。

STEP 11 编译应用程序，结果界面如图7-49所示。

图 7-49　音频采集应用程序

单击“打开文件”按钮，用户选择要保存的音频文件名，确定后，文件名显示在左边的 CEdit 编辑框中，同时用户也可以直接在该编辑框中输入文件路径和文件名。单击“开始采集”按钮，开始采集音频并保存到文件，单击“停止采集”按钮则停止采集音频和保存任务。

STEP 12 添加关闭虚拟消息处理函数，单击对话框资源，单击鼠标右键，选择“属性”命令，单击“消息”按钮。对 WM_CLOSE 消息添加消息处理函数，如图 7-50 所示。

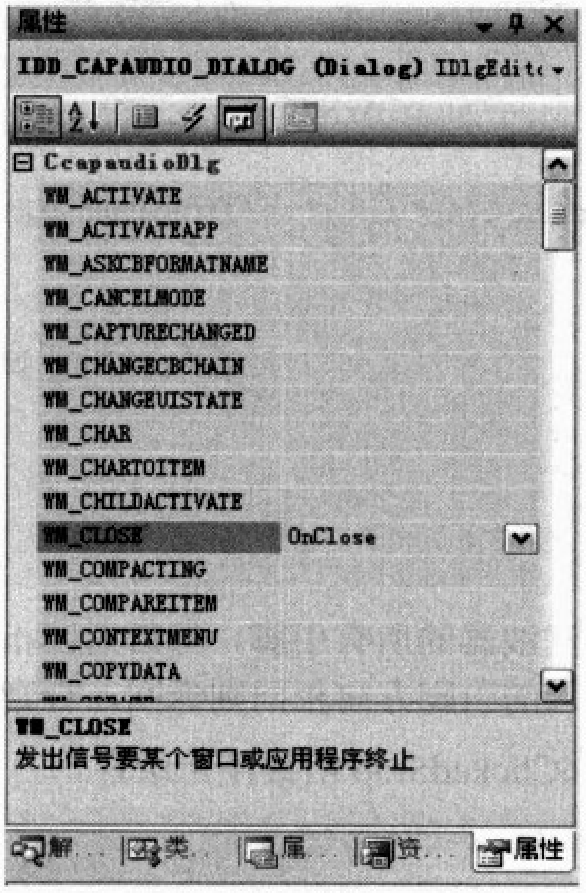


图 7-50　对话框属性的消息资源

代码实现如下。

```
void CcapaudioDlg::OnClose()
{
//TODO: 在此添加消息处理程序代码和/或调用默认值
OnBnClickedStop();   //停止采集、写文件
CoUninitialize();    //反初始化COM库
CDialog::OnClose();
}
```

程序考虑到正在采集、写文件，用户关闭程序时需要先停止采集、写文件操作，然后反初始化 COM 库，最后安全退出应用程序。

7.5 本章小结

DirectShow 是 DirectX 的一个重要组成部分。基于 DirectShow SDK 可以快速开发流媒体的应用程序，支持各种多媒体格式，支持各种编辑操作。VC++ 2005 以其功能强大、界面友好等特点，成为当今软件开发工具的优秀选择。DirectShow SDK 的主要组成部分是滤波器，通过滤波器链表管理这些滤波器。通过各种常见的 DirectShow 开发技术案例：媒体播放案例、消息处理方法、视频采集技术案例、音频采集案例等，剖析了技术设计过程，给出设计流程。

第 8 章

打造自己的媒体播放器

媒体播放器在流媒体开发、应用中经常被提及、使用，它以对流媒体的播放、显示和简单的编辑为主要任务。DirectShow SDK 以其强大的流媒体支持功能，为我们开发流媒体应用程序带来了极大的方便，开发出的媒体播放器既有基本的媒体控制又有自己的特色媒体应用功能。这一节我们就以一个实用案例展示基于 DirectShow SDK 媒体播放器的设计和开发过程，由具有基本功能的播放器，逐步添加其他功能以增强播放器。此开发案例大家既可以直接拿来应用，也可以在此基础上根据需求设计出有自己的特色媒体播放器。

本章重点

- 系统分析与设计
- 实现媒体控制类
- 实现一个简单的媒体播放器
- 升级你的播放器
- 系统效果展示
- 如何播放更多的文件格式

8.1 系统分析与设计

多媒体播放器的功能就是对存储在硬盘上的媒体文件进行回放和显示，在回放或显示过程中，用户可以控制回放的动作特性和显示特性。多媒体播放器案例除了基本的回放和显示动作外，还具备回放速度控制、全屏控制、音量控制、位于顶部显示、拖曳播放和抓图存盘等功能。另外系统还提供了友好的功能按钮信息提示（Tooltip 功能）。

媒体播放器可以播放各种格式的媒体文件（系统必须安装相关流媒体格式的解码库、插件），如 AVI、ASF、MPG、WMA 等常见格式媒体文件。该多媒体播放就具备了典型媒体播放器的功能。使用 DirectShow SDK 开发的媒体播放器可以不用关心媒体文件的格式，只要安装了解码插件集合如 ffmpeg、ffdshow、xvid、divx 等多媒体解码插件，SDK 就会自动剖析、渲染媒体文件，定位媒体文件到对应的解码库解码回放流媒体文件。

多媒体播放器除了“开始/播放/暂停/停止文件”基本功能外，还具有 1/2、1、2 倍速度控制播放，抓图并存盘 BMP 文件，全屏显示图像，静音开关控制，显示置顶开关，播放位置定位和音量控制等功能。

媒体播放器的主画面和主要功能如图 8-1 所示，图中各部分解释如表 8-1 所示。

使用 DirectShow SDK 开发流媒体播放器，主要根据播放的流程，插入、连接和管理各种滤波器，滤波器根据自己的特点完成数据的处理，从而由整个滤波器链表完成流媒体的播放等操作。所以播放器的主要功能就是控制和管理这些滤波器的状态，控制媒体数据流的流向及状态。

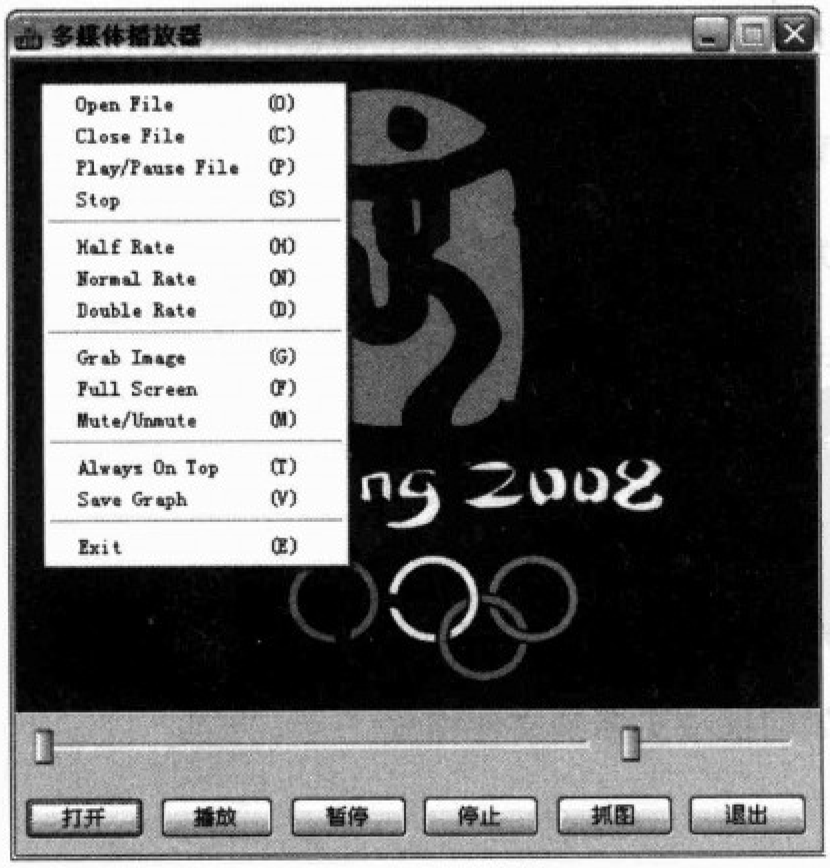


图 8-1　多媒体播放器

表 8-1 媒体播放器部分解释说明

Open File	打开预播放的媒体文件
Close File	关闭正在播放的媒体文件
Play/Pause File	播放/暂停媒体文件
Stop	停止播放媒体文件
Normal Rate	正常速度播放媒体文件
Half Rate	1/2 速度播放媒体文件
Double Rate	2 倍速播放媒体文件
Grab Image	抓图并存盘为 bmp 文件
Full Screen	全屏显示图像，按“Esc”键返回
Mute/UnMute	静音开关，声音有无控制
Always on Top	显示置顶开关，播放时是否总是显示
Save Graph	保存当前文件播放的 Graph 滤波器链表
Exit	退出多媒体播放器系统

8.1.1 FilterGraph结构设计

滤波器链表管理器（Filter Graph Manager）控制和管理插入的各种滤波器，判断其引脚（Pin）的方向和媒体类型，根据滤波器的类型决定是否可以连接。多媒体播放器的一大特点是播放时我们预先并不清楚媒体格式，也不知道该调用哪种解码器。但是，DirectShow 可以通过 IGraphBuilder 接口的 RenderFile 方法，自动剖析媒体格式并连接系统中的解码器。接着，就可以使用 IMediaControl 控制流媒体的播放、暂停和停止。IMediaEventEx 控制媒体消息和状态；IMediaSeeking 控制媒体播放位置和速度。以上所有这些滤波器及接口就是利用 IGraphBuilder 接口作为滤波器链表管理器（Filter Graph Manager）的。

8.1.2 GraphEdit模拟实现

我们在前面已经知道 GraphEdit 管理滤波器的插入、连接和运行，以快速构建一个使用 DirectShow 的滤波器链表。这一章我们设计使用 DirectShow 的多媒体播放器，首先借助 GraphEdit 模拟实现，验证插入的滤波器的运行特点，体会滤波器的操作过程。

现在以回放 C:\ Example.avi 文件为例，使用该工具软件完成媒体回放，具体步骤如下。

STEP 01 启动 GraphEdit.exe，选择“开始”→“程序”→“Microsoft DirectX 9.0 SDK”→“DirectX Utilites”→“GraphEdit”命令，如图 8-2 所示。

STEP 02 单击工具栏中的 按钮，开启“Which filters do you want to insert?”对话框，如图 8-3 所示。

STEP 03 单击“DirectShow Filters”前的加号，展开所属子项，如图 8-4 所示。选择“File

Source（Async）”子项，单击“Insert Filters”按钮插入源滤波器。

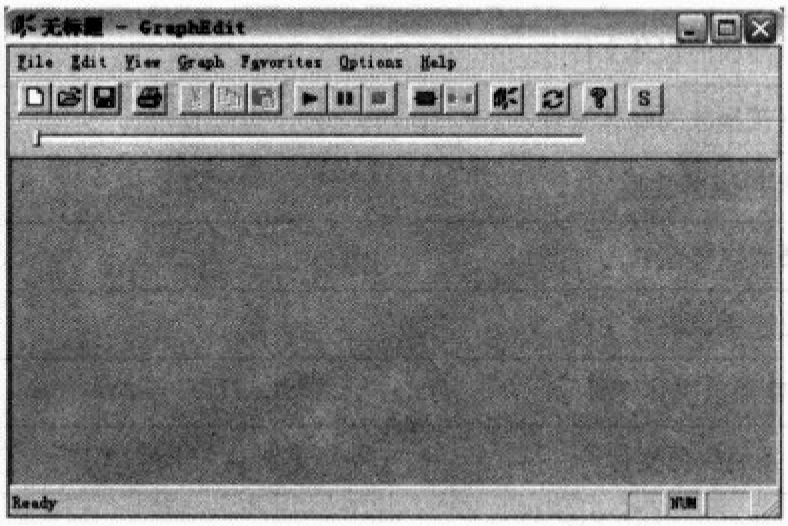


图 8-2　GraphEdit 工具

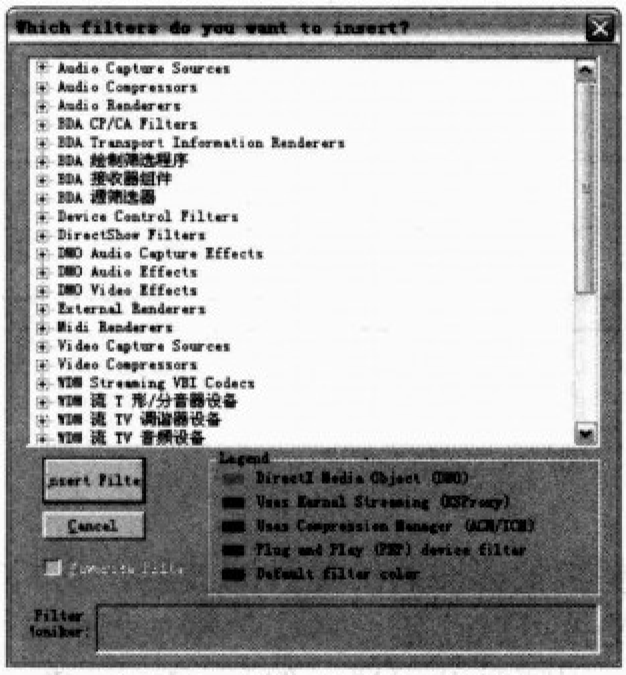


图 8-3　滤波器管理对话框

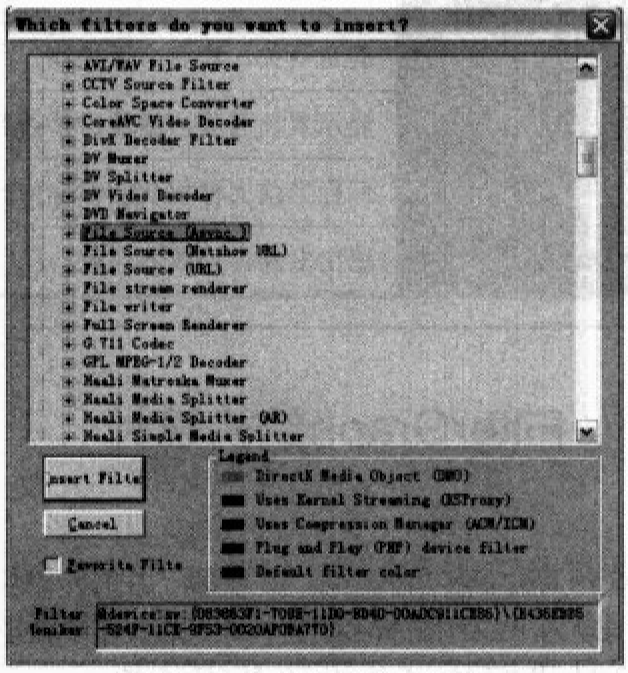


图 8-4　插入源滤波器

STEP 04 在第 3 步中，单击了“Insert Filters”按钮后，会弹出选择文件对话框，如图 8-5 所示。选择预回放的 AVI 文件，这里选择的是“Example.avi”。然后单击“打开”按钮。

STEP 05 单击图 8-4 中“DirectShow Filters”下的“Video Renderer”子项，如图 8-6 所示。

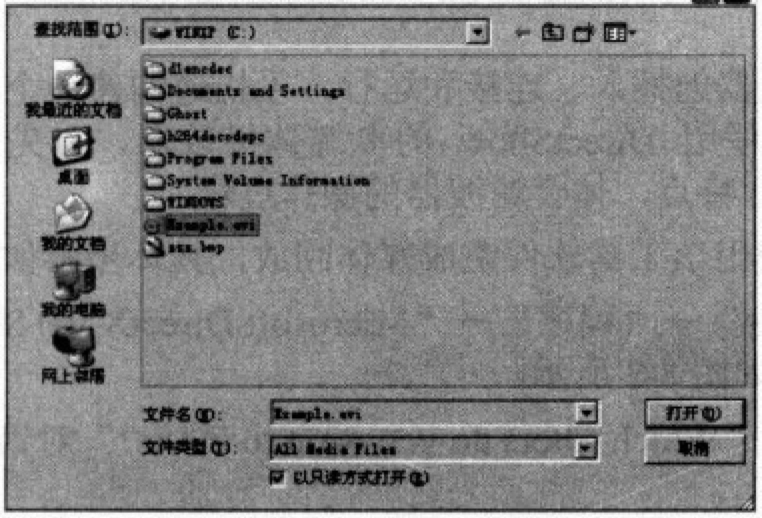


图 8-5　选择播放文件

图 8-6　选择视频渲染滤波器

STEP 06 确保选中“Video Renderer”子项，单击图 8-6 中的“Insert Filters”按钮，结果如图 8-7 所示。

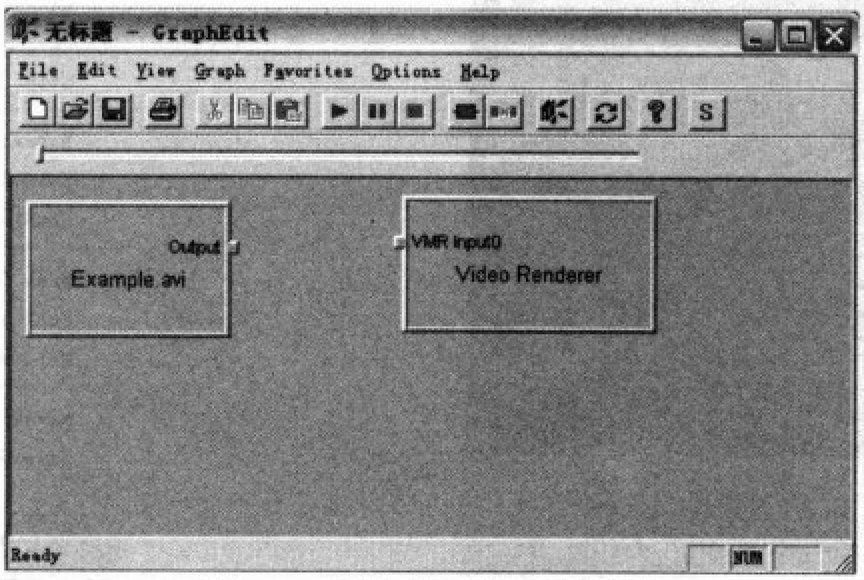


图 8-7 插入滤波器后的界面

STEP 07 最后链接滤波器的引脚，“Output”引脚接“VMR Input0”引脚，如图 8-8 所示。我们发现，由原先的两个滤波器增加了“AVI Spliter”和“Color Space Converter”滤波器，实际上这个过程就是 DirectShow 的接口自动判断媒体类型，添加滤波器而智能链接的。

STEP 08 至此，就完成了构建回放 Example.avi 文件的滤波器链表。单击工具栏中的 ► 按钮，运行链表，结果如图 8-9 所示。

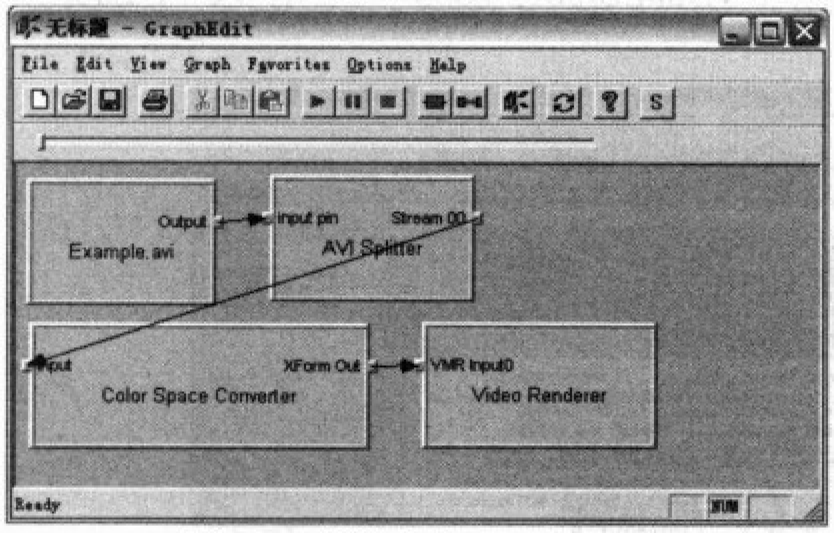


图 8-8 链接滤波器引脚图

图 8-9 回放 Example.avi

8.1.3 界面设计

我们设计的流媒体播放器的主界面类似于操作系统的传统媒体播放器（MediaPlayer）。以对话框的形式设计主界面，窗口内支持右键快捷菜单，设计后的主界面如图 8-10 所示，菜单内容如图 8-11 所示。

在界面设计中，使用 VC++ 2005 的项目“资源视图”和“属性”页，对添加的控件的“Caption”和“ID”进行更改和设置。激活“属性”页的途径：用鼠标右键单击对应控件，在弹出的快捷菜单中选择“属性”命令。

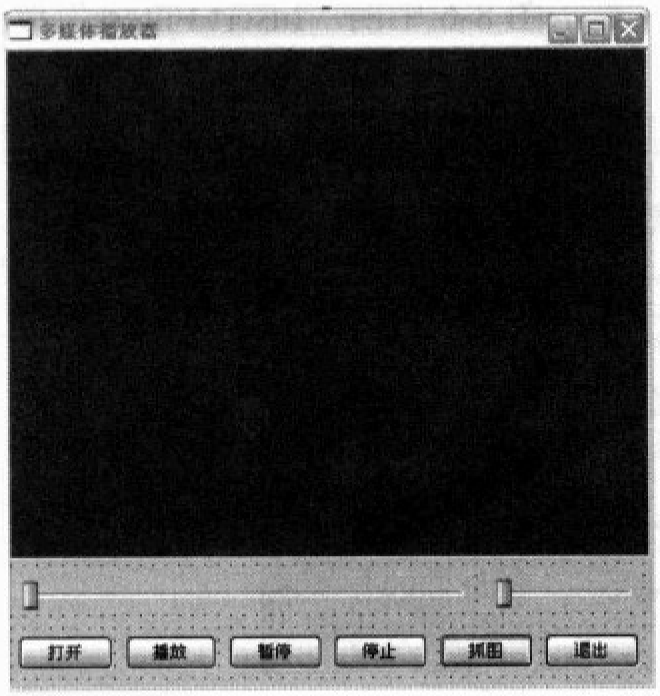


图 8-10 主界面控件设计

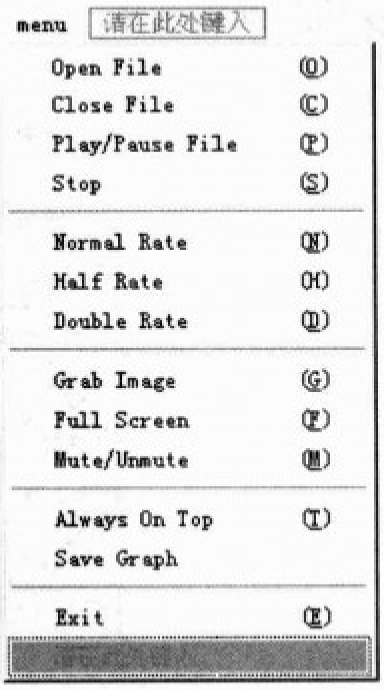


图 8-11 主菜单内容设计

1．创建对话框应用程序

我们设计的多媒体播放器以对话框为基础，以按钮和菜单实现流媒体控制。下面我们从建立对话框应用程序开始，添加控件、菜单。创建对话框应用程序的过程如下。

STEP 01 单击桌面上的快捷方式“Microsoft Visual Studio 2005”，启动 VC++ 2005，开启如图 8-12 所示的画面。

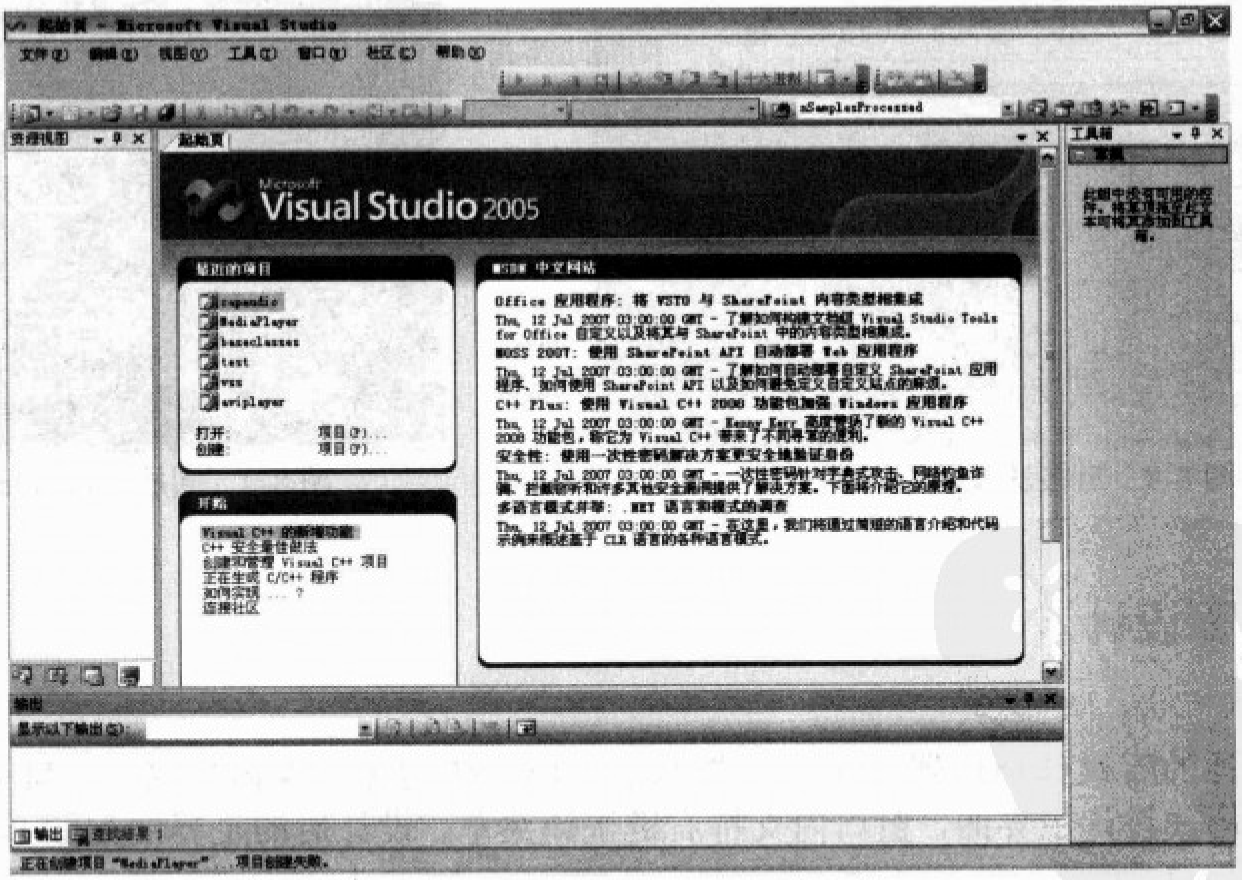


图 8-12 VC++ 2005 集成开发环境

STEP 02 选择“文件”→“新建”→“项目”命令，启动“新建项目”对话框，如图 8-13 所示。

STEP 03 选择图 8-13 中的“项目类型”中的“Visual C++”选项下的“MFC”子项，然后选中右边的模板“MFC 应用程序”。将项目命名为 MediaPlayer，确定存放位置。然后单击

"确定"按钮，出现图 8-14 所示的画面，这样就启动了"MFC 应用程序向导"。

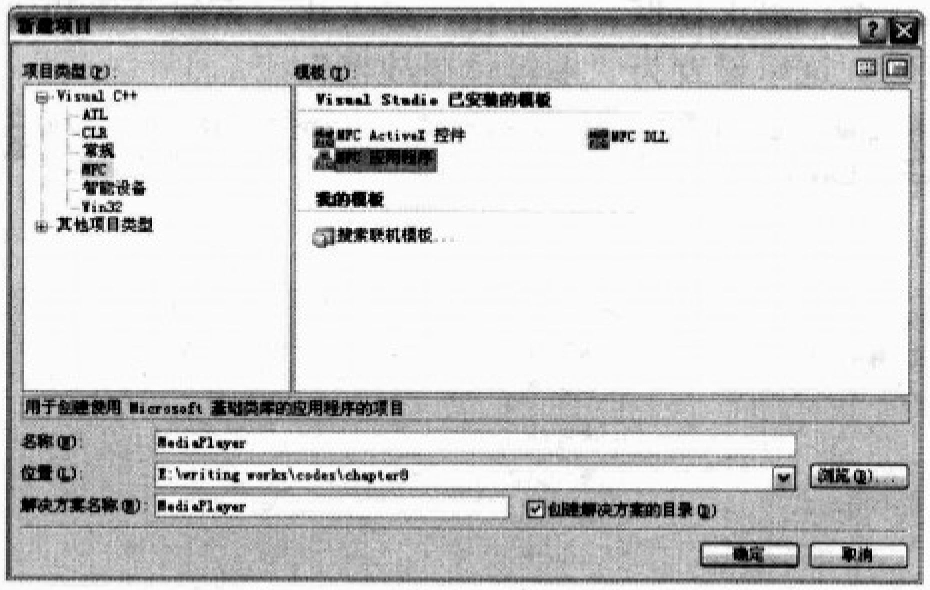


图 8-13 "新建项目"对话框

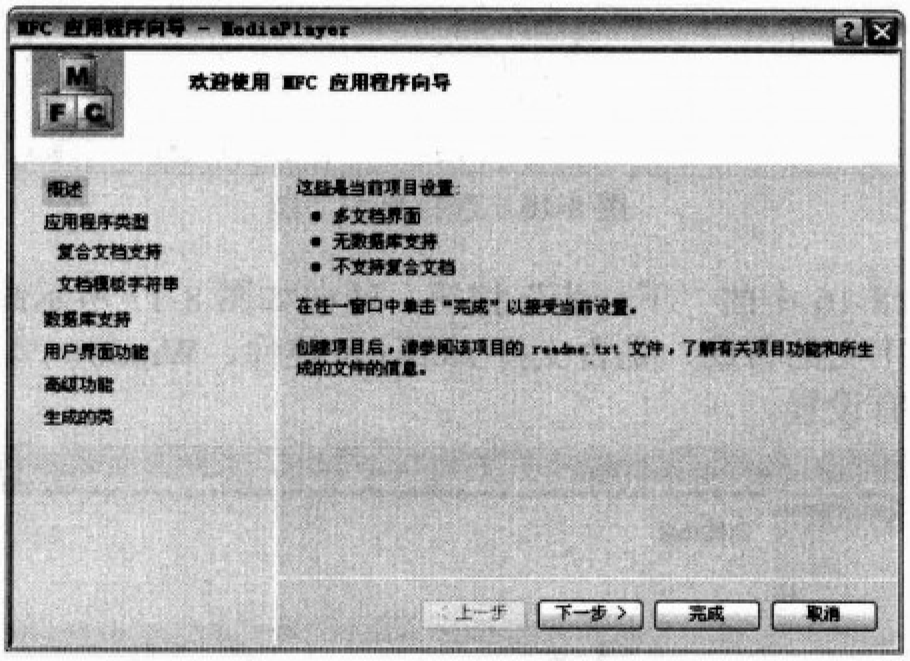


图 8-14 MFC 应用程序向导

STEP 04 单击图 8-14 中的"下一步"按钮，转向如图 8-15 所示画面，用户选择应用程序类型。VC++ 2005 支持单文档、多个文档和对话框应用程序，选择 MFC 的使用方式，选择程序是否使用 Unicode 库。这里只改变应用程序的类型为"基于对话框"，其他为默认值。

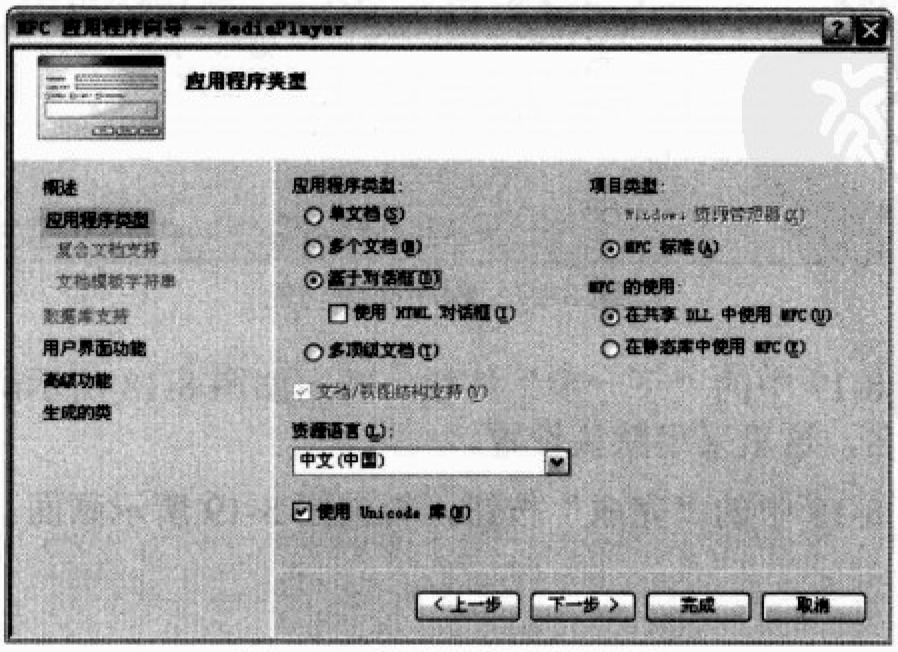


图 8-15 选择应用程序类型

STEP 05 单击图 8-15 中的“下一步”按钮，转向图 8-16 所示画面，用户选择界面功能。主框架样式包括最小化框、最大化框、最小化、最大化、系统菜单和“关于”框，用户可以在此设置对话框的标题。这里设置为“多媒体播放器”。

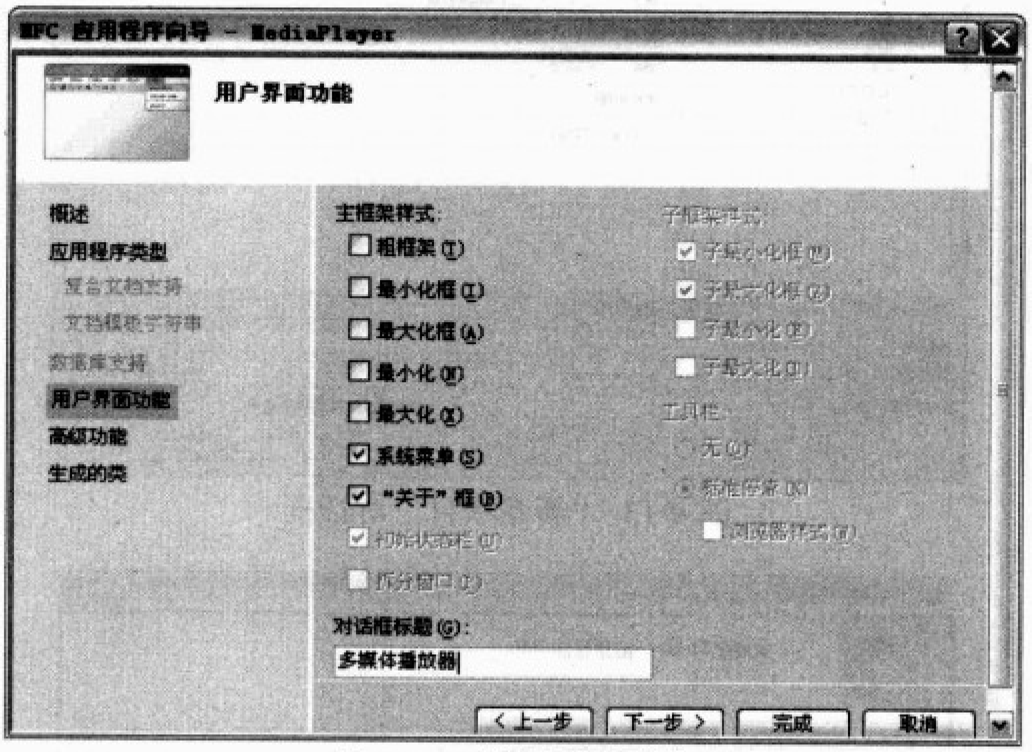


图 8-16　选择界面功能

STEP 06 单击图 8-16 中的“下一步”按钮，转向如图 8-17 所示画面，选择程序高级功能。这些功能包括上下文的帮助、是否支持 ActiveX 控件、Windows 套接字（即网络支持）等功能。这里默认所有设置。

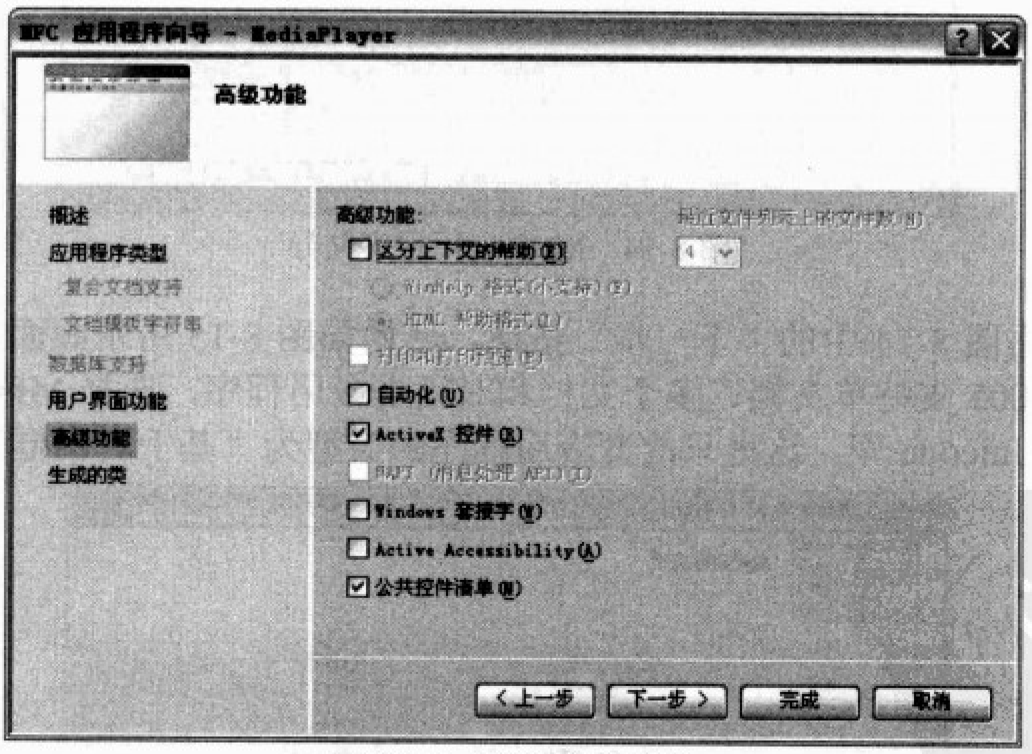


图 8-17　程序高级功能

STEP 07 单击图 8-17 中的“下一步”按钮，转向如图 8-18 所示画面，设置向导生成的类。用户可以更改类名，这里采用默认设置。

STEP 08 单击图 8-18 中的“完成”按钮，转向图 8-19 所示画面。至此建立了对话框应用程序的基本框架。

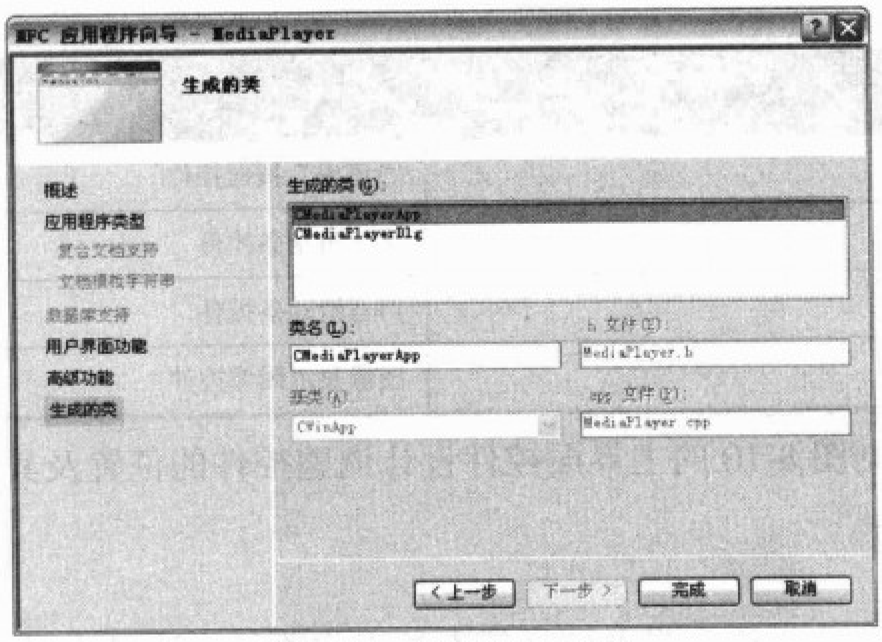


图 8-18　选择生成的类

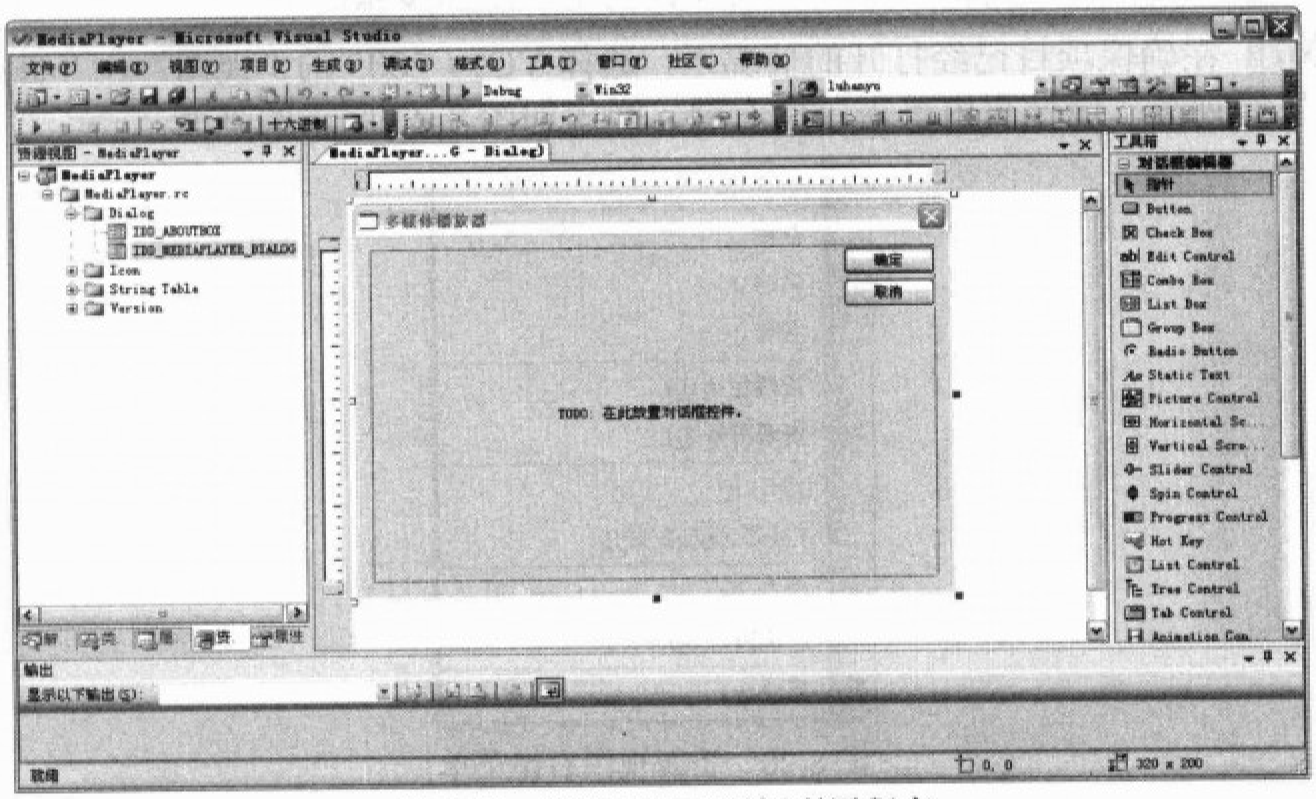


图 8-19　创建后的对话框应用程序

2. 修改对话框

添加 6 个按钮控件 Button、2 个滑动条控件 Slider Control 和 1 个图像控件 Picture Control。控件清单如表 8-2 所示。

表 8-2　控件列表

控件	说明
DC_BUTTON_OPEN	“打开”按钮控件
IDC_BUTTON_PLAY	“播放”按钮控件
IDC_BUTTON_PAUSE	“暂停”按钮控件
IDC_BUTTON_STOP	“停止”按钮控件
IDC_BUTTON_GRASP	“抓图”按钮控件

（续表）

控件	说明
IDC_BUTTON_EXIT	“退出”按钮控件
IDC_SLIDER_PLAY	播放滑动条控件
IDC_SLIDER_VOLUME	声音滑动条控件
IDC_VIDEO_WINDOW	图像显示图像控件

添加完控件，根据图 8-10 的主界面控件设计调整控件的位置及其大小。

3. 添加菜单资源

菜单在应用程序中是非常重要也是最基本的操作路径。这里为对话框应用程序添加菜单支持。在上述的应用程序的框架下，下列步骤列出了添加步骤。

STEP 01 在确保项目已经打开的情况下，选择 VC++ 2005 的“视图”→“资源视图”命令，在“资源视图”属性页内单击鼠标右键，弹出如图 8-20 所示的画面。

图 8-20　资源视图右键菜单

STEP 02 选择图 8-20 中的“添加资源”命令，激活添加资源对话框，如图 8-21 所示。资源类型包括位图（Bitmap）、光标（Cursor）、对话框（Dialog）、图标（Icon）、菜单（Menu）、字符串表（String Table）、工具栏（Toolbar）和版本信息（Version）。这些资源既可以新建，也可以从已有的资源中导入或自定义。

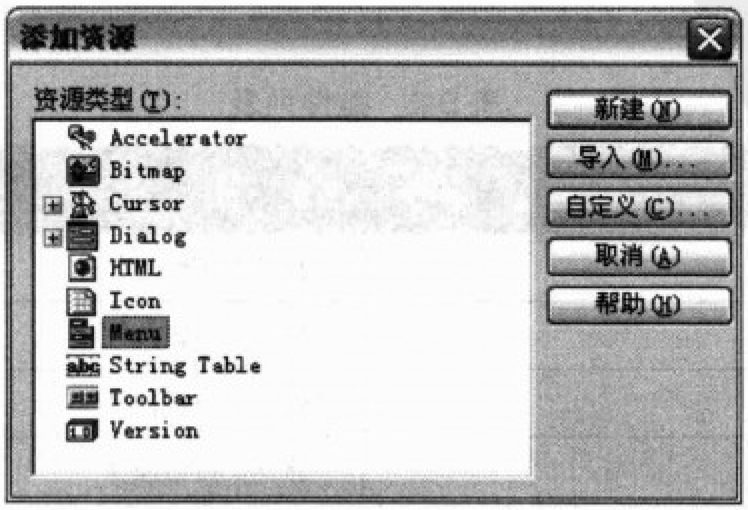


图 8-21　“添加资源”对话框

STEP 03 单击图 8-21 中的“Menu”子项，然后单击“新建”按钮，启动如图 8-22 所示的画面。

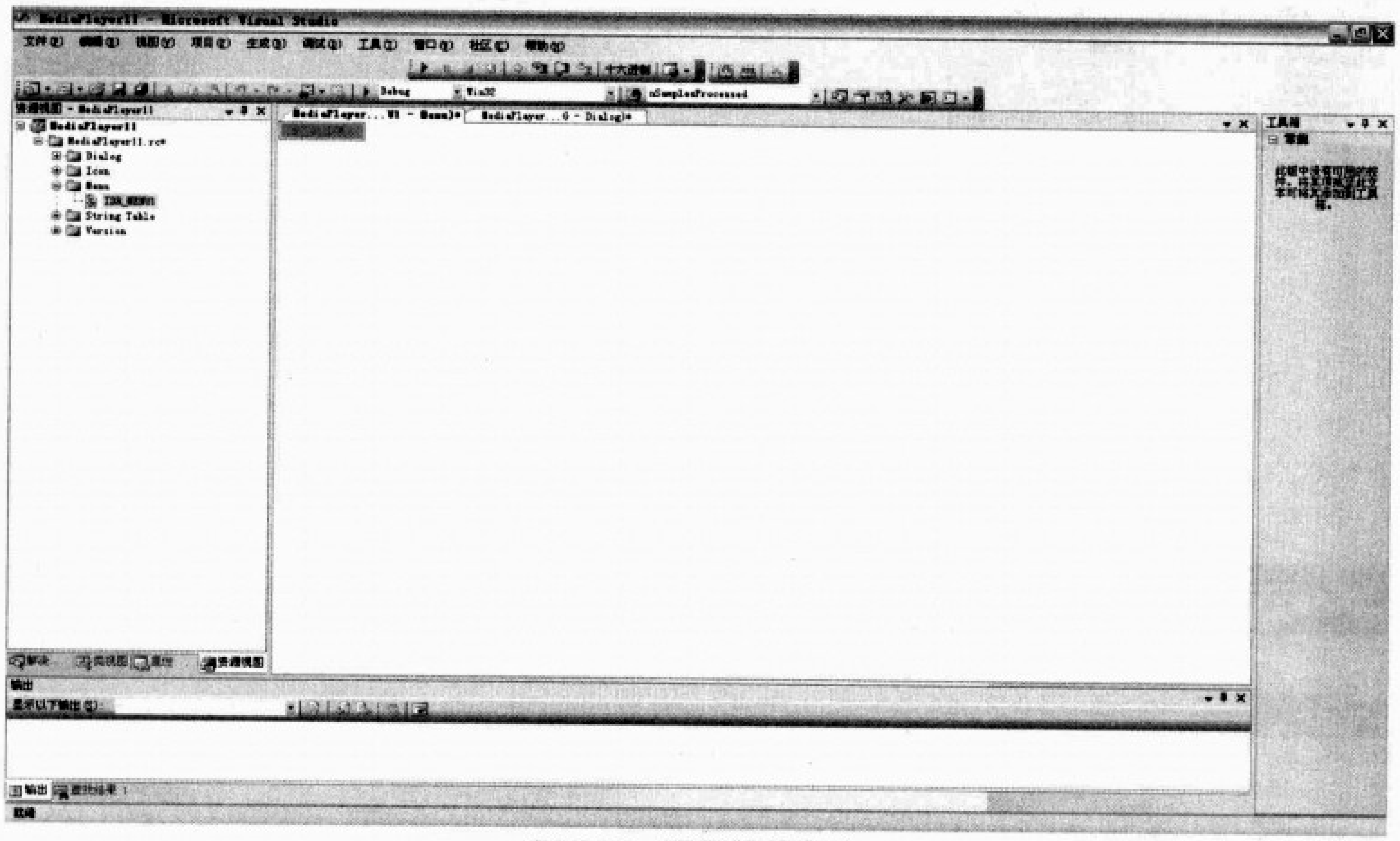


图 8-22　菜单编辑状态

STEP 04 单击图 8-22 中的“请在此键入”，输入菜单内容，如图 8-23 所示。依次根据表 8-3 中的菜单内容添加菜单。

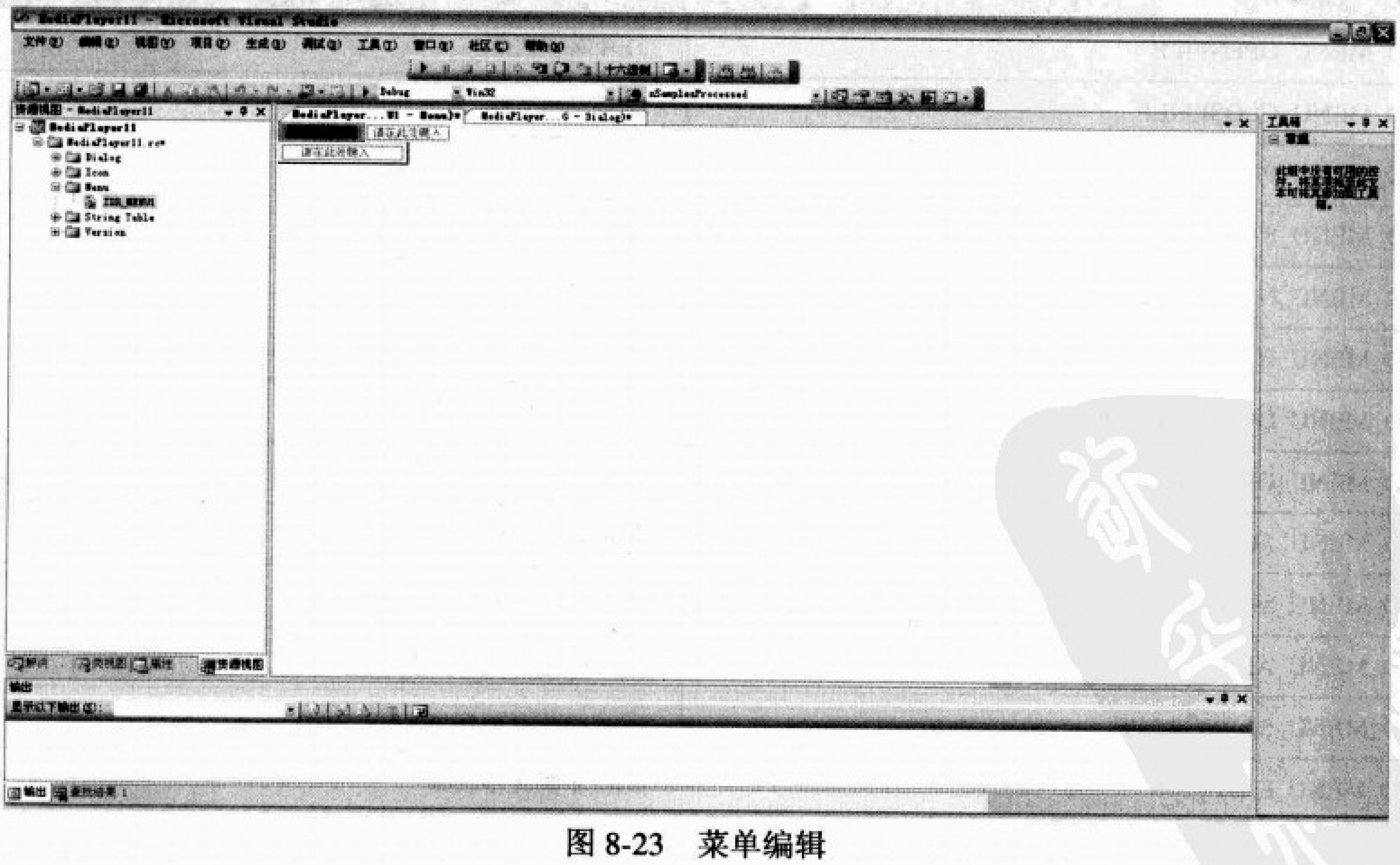


图 8-23　菜单编辑

添加完成的菜单资源如图 8-24 所示。菜单的每个子项右对齐，且有快捷方式，如：Open File\t(&O)，\t 表示空出制表 tab 大小，(&O)表示快捷方式。

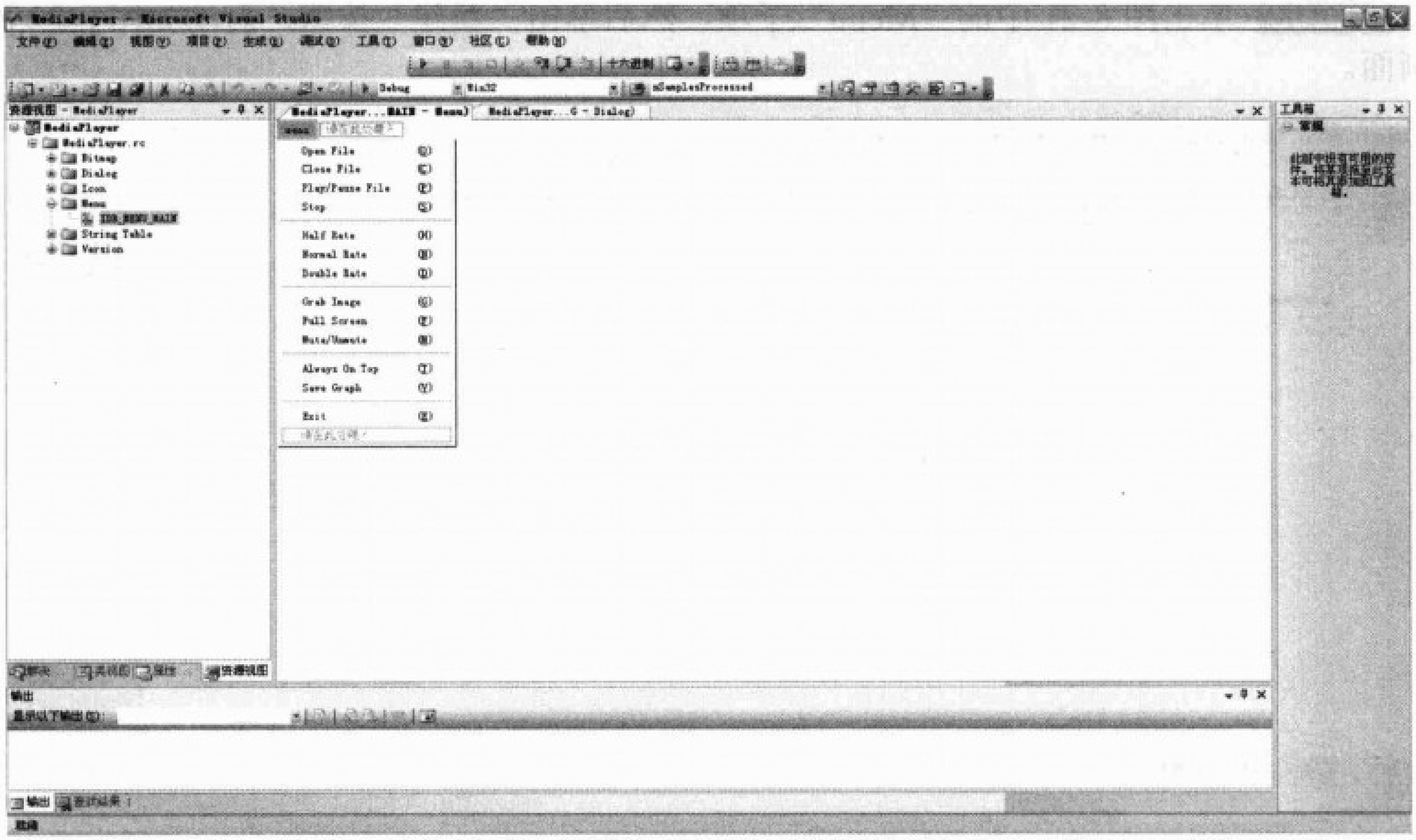


图 8-24 完成的菜单编辑

MediaPlayer 应用程序的菜单列表如表 8-3 所示。

表 8-3 菜单列表

菜单	说明
ID_MENU_OPENFILE	打开媒体文件
ID_MENU_CLOSEFILE	关闭媒体文件
ID_MENU_PLAY	播放媒体文件
ID_MENU_STOP	停止播放文件
ID_MENU_NORMALRATE	正常速率播放
ID_MENU_HALFRATE	1/2 速率播放文件
ID_MENU_DOUBLERATE	双倍速率播放文件
ID_MENU_GRABIMAGE	抓图并存盘
ID_MENU_FULLSCREEN	全屏显示图像
ID_MENU_MUTE	静音开关控制
ID_MENU_ALWAYSONTOP	窗口总在最顶端显示
ID_MENU_SAVEGRAPH	保存滤波器链表为文件
ID_MENU_EXIT	退出多媒体播放器

4．添加位图资源

在如图 8-25 所示的“添加资源”对话框中，选中“Bitmap”选项，然后单击“导入”按

钮，在弹出的对话框中选择一个图片文件并将其打开。命名其“ID”为“IDB_BITMAP_BKGROUND”，将该图片作为背景欢迎图片，如图 8-26 所示。

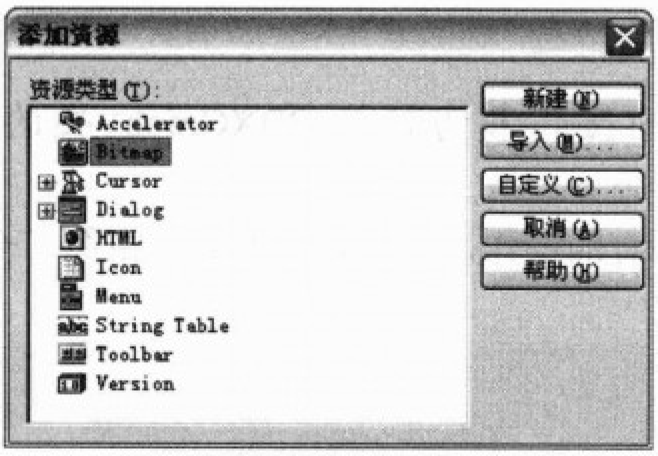


图 8-25 “添加资源”对话框

图 8-26 BMP 位图资源

编译链接设计应用程序，运行后的多媒体播放器程序如图 8-27 所示。在图像显示区域单击鼠标右键可弹出快捷菜单，菜单的子项包括应用程序的按钮功能。打开媒体文件、播放/暂停/停止媒体文件、抓图等。主界面还包括媒体播放进度的滑动条，且支持任意拖动。支持音量控制功能。

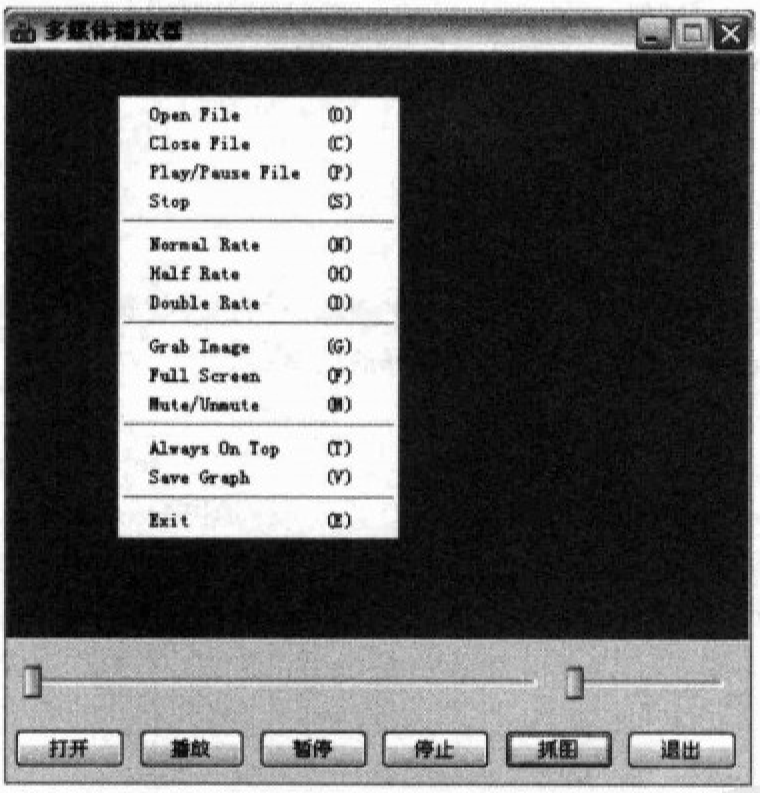


图 8-27 多媒体播放器

8.2 实现媒体控制类

DirectShow SDK 对开发者来说是一套库函数，使用过程大体来说是先创建，然后使用，最后销毁，媒体的播放控制和图像显示等动作都是由 SDK 完成的。为了系统开发方便、易升级和功能扩展，作者把有关 SDK 的功能开发函数封装在类 CDXGraph 中。应用程序使用其中的成员函数和变量完成媒体的播放及显示，该类包含了回放媒体时需要的几乎所有的动作和控制方法，该方案使用了大部分成员函数，还有少量方法函数没有使用。开发者可以在该案例的基础上，使用该类构建出功能更强大、回放控制更灵活的多媒体播放器。

8.2.1 CDXGraph类初始化

CDXGraph 类封装了有关 DirectShow 的接口和方法。CDXGraph 类的实现在 CDXGraph.cpp 中，头文件为 CDXGraph.h。

CDXGraph 类为播放媒体文件准备了几乎所有的操作，包括音频控制、抓图、全屏和任意定位等高级功能。具体代码如下。

```
class CDXGraph
{
public:
 IGraphBuilder   *pGraph;            //滤波器链表管理器
 IMediaControl   *pMediaControl;     //媒体控制接口，如run、stop、pause
 IMediaEventEx   *pMediaEvent;       //媒体事件接口
 IBasicVideo         *pBasicVideo;   //视频基本接口
 IBasicAudio         *pBasicAudio;   //音频基本接口
 IVideoWindow    *pVideoWindow;      //视频窗口接口
 IMediaSeeking   *pMediaSeeking;     //媒体定位接口
public:
 CDXGraph();
virtual ~CDXGraph();
public:
 virtual bool Create(void);      //生成滤波器链表管理器
 virtual void Release(void);     //释放所有接口
 virtual bool Attach(IGraphBuilder * inGraphBuilder);
 //IGraphBuilder * GetGraph(void);        //Not outstanding reference count
 IMediaEventEx * GetEventHandle(void);    //返回IMediaEventEx指针
 //根据引脚方向连接滤波器
 bool ConnectFilters(IPin * inOutputPin, IPin * inInputPin, const AM_MEDIA_TYPE
* inMediaType = 0);
 //断开连接滤波器
 void DisconnectFilters(IPin * inOutputPin);
 //设置显示窗口
 bool SetDisplayWindow(HWND inWindow);
 //设置窗口通知消息
 bool SetNotifyWindow(HWND inWindow);
 //窗口大小改变处理函数
 bool ResizeVideoWindow(long inLeft, long inTop, long inWidth, long inHeight);
 //处理事件
 void HandleEvent(WPARAM inWParam, LPARAM inLParam);
 //媒体运行状态
 bool Run(void);          //控制滤波器链表，运行媒体
 bool Stop(void);         //停止媒体
 bool Pause(void);        //暂停媒体
 bool IsRunning(void);    //媒体是否在运行
```

```
    bool IsStopped(void);     //媒体是否已停止
    bool IsPaused(void);      //媒体是否已暂停
                              //设置显示窗口全屏显示
    bool SetFullScreen(BOOL inEnabled);
    bool GetFullScreen(void);
    //媒体定位
    bool GetCurrentPosition(double * outPosition);                  //获取当前播放位置
    bool GetStopPosition(double * outPosition);                     //获取停止位置
    bool SetCurrentPosition(double inPosition);                     //设置当前位置
    bool SetStartStopPosition(double inStart, double inStop);  //设置开始和停止位置
    bool GetDuration(double * outDuration);                         //获取文件时间长度
    bool SetPlaybackRate(double inRate);                            //设置回放速率
   //设置媒体音量: range from -10000 to 0, and 0 is FULL_VOLUME
    bool SetAudioVolume(long inVolume);
    long GetAudioVolume(void);
    //设置音频平衡: range from -10000(left) to 10000(right), and 0 is both
    bool SetAudioBalance(long inBalance);
    long GetAudioBalance(void);
    //剖析媒体文件
    bool RenderFile(const char * inFile);
    bool SnapshotBitmap(const char * outFile);
    int m_nVolume;
    void ChangeAudioVolume(int nVolume);
    //静音开关
    void Mute();
    void UnMute();
  private:
    //供GraphEdit调试时使用
    void AddToObjectTable(void) ;
    void RemoveFromObjectTable(void);
    //查询有关接口
    bool QueryInterfaces(void);
  };
```

类的构造器中将所有的指针清零。同时，MediaPlayer.cpp 文件的 InitInstance()函数中对 COM 初始化：CoInitialize(NULL);。

8.2.2 创建Graph滤波器链表

根据第 7 章介绍的使用 DirectShow 开发流媒体应用程序，首先创建滤波器链表管理器，然后在该链表下查询、使用各种接口，包括媒体控制接口、媒体事件接口、基类音频/视频接口、视频窗口接口、媒体定位接口等。

创建滤波器链表管理 Graph 图的代码如下：

```
  bool CDXGraph::Create(void)
  {
```

```
    if (!pGraph)              //pGraph为空，则创建
    { //创建滤波器链表管理器pGraph
    if (SUCCEEDED(CoCreateInstance(CLSID_FilterGraph,
                                    NULL,
                                   CLSCTX_INPROC_ SERVER,
                                    IID_IGraphBuilder, (void **)&pGraph)))
        {
            AddToObjectTable();  //添加到目标列表以使用GraphEdit调试
             //访问pGraph中的有关接口
            return QueryInterfaces();
        }
        pGraph = 0;
    }
    return false;
}
```

上述程序中，如果滤波器链表不为空，即再次调用该函数则不再重复创建。若为空，则创建链表管理器pGraph，然后把pGraph添加到目标列表以便于使用GraphEdit程序进行调试。最后基于pGraph查询各种接口。

在滤波器链表管理器pGraph中，查询、初始化必需的DirectShow接口，其代码如下。

```
bool CDXGraph::QueryInterfaces(void)
{
 if (pGraph) //pGraph不为空。
 {
        HRESULT hr = NOERROR;  //函数返回值初始化
        //查询媒体控制接口
        hr |=pGraph->QueryInterface(IID_IMediaControl,(void**)&pMediaControl);
        //查询媒体事件接口
        hr |= pGraph->QueryInterface(IID_IMediaEventEx, (void **)&pMediaEvent);
        //查询基类视频接口
        hr |= pGraph->QueryInterface(IID_IBasicVideo, (void **)&pBasicVideo);
        //查询基类音频接口
        hr |= pGraph->QueryInterface(IID_IBasicAudio, (void **)&pBasicAudio);
        //查询视频窗口接口
        hr |= pGraph->QueryInterface(IID_IVideoWindow, (void **)&pVideoWindow);
        //查询媒体定位接口
        hr |=pGraph->QueryInterface(IID_IMediaSeeking,(void**)&pMediaSeeking);
        if (pMediaSeeking)
        {   //查询媒体定位接口成功，则设置时间格式
            pMediaSeeking->SetTimeFormat(&TIME_FORMAT_MEDIA_TIME);
        }
        //返回以上操作的结果
        return SUCCEEDED(hr);
 }
 return false;
}
```

上述程序中，在滤波器管理器不为空的情况下，查询各种必需的接口。每次操作的结果放置在 hr 中（返回值操作或运算），最后判断 hr 的值。

8.2.3 图像窗口设计

首先打开媒体文件，渲染、剖析其媒体格式并连接对应解码器。

```
bool CDXGraph::RenderFile(const char * inFile)
{
 if (pGraph)
 {
        WCHAR    szFilePath[MAX_PATH];
        //把传入的文件名转换成宽字符串
        MultiByteToWideChar(CP_ACP, 0, inFile, -1, szFilePath, MAX_PATH);
        if (SUCCEEDED(pGraph->RenderFile(szFilePath, NULL)))
        {   //渲染媒体文件，构建滤波器链表，若成功则返回真值。
              return true;
        }
 }
 return false;
}
```

上述程序中，由于传入的文件名是单字符串（一个字符占用一个字节），但是 DirectShow SDK 底层使用的是宽字符串，所以需要首先进行转换，然后调用链表管理器的方法 RenderFile，自动剖析媒体格式，选择对应的滤波器并连接起来，最后成功则返回真值，否则返回值“0”。

设置媒体显示窗口，把输入窗口句柄与 DirectShow 控制接口 pVideoWindow 捆绑。

```
//输入显示窗口的句柄：inWindow
bool CDXGraph::SetDisplayWindow(HWND inWindow)
{
 if (pVideoWindow)
 {
        //首先隐藏视频窗口
        pVideoWindow->put_Visible(OAFALSE);
        pVideoWindow->put_Owner((OAHWND)inWindow);
            //获取输入窗口的显示区域
        RECT windowRect;
        ::GetClientRect(inWindow, &windowRect);
        pVideoWindow->put_Left(0);
        pVideoWindow->put_Top(0);
        pVideoWindow->put_Width(windowRect.right - windowRect.left);
        pVideoWindow->put_Height(windowRect.bottom - windowRect.top);
pVideoWindow->put_WindowStyle(WS_CHILD|WS_CLIPCHILDREN|WS_CLIPSIBLINGS);
        pVideoWindow->put_MessageDrain((OAHWND) inWindow);
        //恢复视频窗口
```

```
        if (inWindow != NULL) {
              pVideoWindow->put_Visible(OATRUE);
        }else{
              pVideoWindow->put_Visible(OAFALSE);
        }
        return true;
  }
  return false;
}
```

上述程序中的任务是把传入的视频显示窗口与DirectShow的视频窗口接口捆绑。首先隐藏视频窗口，设置视频窗口所属为传入的显示窗口。然后获取传入的显示窗口的区域，并把该区域设置到DirectShow的视频窗口，包括左边、上边的起点坐标、窗口的宽和高、窗口的显示模式及窗口消息的处理方式等。最后恢复视频窗口，完成视频窗口的设置。

设置窗口消息通知，首先在文件CDXGraph.h中自定义消息WM_GRAPHNOTIFY。

```
//滤波器链表通知给特定窗口
#define WM_GRAPHNOTIFY  (WM_USER+20)
```

实现窗口消息、事件通知函数SetNotifyWindow，输入显示窗口句柄inWindow。

```
bool CDXGraph::SetNotifyWindow(HWND inWindow)
{
 if (pMediaEvent)     //媒体事件接口不为空
 {
       //设置消息通知窗口，消息WM_GRAPHNOTIFY为用户定义
       pMediaEvent->SetNotifyWindow((OAHWND)inWindow,
                                        WM_GRAPHNOTIFY, 0);
       return true;  //设置成功
 }
 return false;          //媒体事件接口为空
}
```

上述程序实现消息处理的窗口的设置，媒体的播放等事件以消息的方式通知给主窗口。消息标识为用户自定义。

8.2.4 媒体播放控制

媒体控制包括运行（Run）、暂停（Pause）、停止（Stop），播放媒体的定位和静音开关。以下是媒体播放控制：Run/Pause/Stop的操作代码。

```
/*运行媒体*/
bool CDXGraph::Run(void)
{
 if (pGraph && pMediaControl){    //链表和媒体控制接口均不为空
       if (!IsRunning()){            //判断是否正在运行
              if (SUCCEEDED(pMediaControl->Run())){ //开始运行媒体
```

```
                    return true;        //正在运行
                }
        } else{                         //正在运行
                return true;
        }
    }
 return false;
}
/*停止媒体*/
bool CDXGraph::Stop(void)
{
 if (pGraph && pMediaControl){          //链表和媒体控制接口均不为空
        if (!IsStopped()){              //判断是否已经停止
                if (SUCCEEDED(pMediaControl->Stop()))   { //开始停止媒体
                        return true;    //已经停止
                }
        } else {
                return true;            //已经停止
        }
 }
 return false;
}
/*暂停媒体*/
bool CDXGraph::Pause(void)
{
 if (pGraph && pMediaControl) {         //链表和媒体控制接口均不为空
        if (!IsPaused()) {              //判断是否已经暂停
                if (SUCCEEDED(pMediaControl->Pause())){ //开始暂停媒体
                    return true;        //已经暂停
                }
        } else {
                return true;            //已经暂停
        }
 }
 return false;
}
```

上述程序中，3 个操作的过程基本相同，首先判断链表和媒体控制接口不为空，则判断是否已经处于待检查的状态，如果没有则开始执行，然后返回真值表示成功，否则返回零。

播放媒体的定位，包括文件的时间长度、设置/获取当前播放位置和设置回放速率等。

```
/*获取播放文件的时间长度*/
bool CDXGraph::GetDuration(double * outDuration)
{
 if (pMediaSeeking)  {               //媒体定位接口不为空
        __int64 length = 0;          //媒体长度
        if (SUCCEEDED(pMediaSeeking->GetDuration(&length))){    //获取媒体长度
```

```
            *outDuration = ((double)length) / 10000000.;//返回长度，单位
100ns
            return true;
        }
    }
    return false;
}
/*获取当前播放位置*/
bool CDXGraph::GetCurrentPosition(double * outPosition)
{
    if (pMediaSeeking){                 //媒体定位接口不为空
        __int64 position = 0;           //媒体位置
        //获取媒体长度
        if (SUCCEEDED(pMediaSeeking->GetCurrentPosition(&position))){
            *outPosition = ((double)position) / 10000000.;  //返回位置，单位100ns
            return true;
        }
    }
    return false;
}
/*设置当前播放位置*/
bool CDXGraph::SetCurrentPosition(double inPosition)
{
    if (pMediaSeeking)                  //媒体定位接口不为空
    {
        __int64 one = 10000000; // 1/100ns
        __int64 position = (__int64)(one * inPosition);
        HRESULT hr = pMediaSeeking->SetPositions(&position,
                        AM_SEEKING_AbsolutePositioning//绝对位置
                        AM_SEEKING_SeekToKeyFrame,      //查询直到关键帧
                    0, AM_SEEKING_NoPositioning);       //没有设置停止位置
        return SUCCEEDED(hr);
    }
    return false;
}
/*设置回放速率*/
bool CDXGraph::SetPlaybackRate(double inRate)
{
    if (pMediaSeeking){         //媒体定位接口不为空
        if (SUCCEEDED(pMediaSeeking->SetRate(inRate))){ //设置回放速率
            return true;
        }
    }
    return false;
}
```

上述程序是对文件播放的定位设置。设置的过程基本相同，首先确保媒体定位接口不为

空，然后使用媒体定位接口下的方法实现定位。需要特别注意的是定位的单位为 100ns，而外部的时间是以秒为单位的，注意之间的转换。

另外还有 GetStopPosition、SetStartStopPosition 设置媒体播放位置。

静音开与关的操作代码如下：

```
void CDXGraph::Mute()
{
 if (!pBasicAudio)                              //如果基类音频接口为空则直接返回
      return;
 pBasicAudio->put_Volume(-10000);               //设置静音。原理是把音量设置为最小
}
void CDXGraph::UnMute()
{
 if (!pBasicAudio)                              //如果基类音频接口为空则直接返回
      return;
 long lVolume = (m_nVolume - 100) * 100;  //获取原来的音量大小
 pBasicAudio->put_Volume(lVolume);              //设置音量
}
```

静音开关是通过使用基类音频的方法 put_Volume 完成的，静音是把音量值设置为最小（-10000），关闭静音即恢复到原来的音量。

8.2.5 全屏显示及抓图存盘实现

全屏显示是媒体播放器最基本的功能之一，DirectShow 的视频窗口接口有专门的方法支持全屏显示功能。抓图存盘也是播放器的基本功能，基类视频接口提供获取当前播放的视频帧数据。

全屏显示图像的代码如下：

```
//全屏显示函数开关
bool CDXGraph::SetFullScreen(BOOL inEnabled)
{
 if (pVideoWindow)   //视频窗口接口不为空
 {
    //设置全屏显示是个开关，用true和false表示开和关。
HRESULT hr = pVideoWindow->put_FullScreenMode(inEnabled ? OATRUE : OAFALSE);
     return SUCCEEDED(hr);
 }
 return false;
}
```

上述程序传入 BOOL 值表示是否设置为全屏显示，函数 put_FullScreenMode 是视频窗口接口的方法。

对正在播放的视频快照抓图并存盘，代码如下。

```
//视频快照
```

```
bool CDXGraph::SnapshotBitmap(const char * outFile)
{
 if (pBasicVideo)                    //基类视频接口不为空
 {
       long bitmapSize = 0;          //位图大小
       //仅获得当前图像的位图大小
       if (SUCCEEDED(pBasicVideo->GetCurrentImage(&bitmapSize, 0)))
       {
              bool pass = false;  //获取成功标识
              unsigned char * buffer = new unsigned char[bitmapSize]; //缓冲区
              //以bitmapsize大小读取图像数据，内容放在buffer中。
       if (SUCCEEDED(pBasicVideo->GetCurrentImage(&bitmapSize, (long *)buffer)))
              {
                BITMAPFILEHEADER hdr;           //位图文件头
                LPBITMAPINFOHEADER    lpbi;    //位图文件信息头
                lpbi = (LPBITMAPINFOHEADER)buffer;
                int nColors = 0;
                //以下为创建位图的文件头结构体字段信息
                if (lpbi->biBitCount < 8)
                {
                       nColors = 1 << lpbi->biBitCount;
                }
                hdr.bfType         = ((WORD) ('M' << 8) | 'B');    //always is "BM"
                hdr.bfSize         = bitmapSize + sizeof( hdr );
                hdr.bfReserved1    = 0;
                hdr.bfReserved2    = 0;
                hdr.bfOffBits      = (DWORD) (sizeof(BITMAPFILEHEADER) + lpbi->
biSize + nColors * sizeof(RGBQUAD));
                //以可读、写二进制创建文件
    CFile bitmapFile(LPCTSTR(outFile), CFile::modeReadWrite | CFile::modeCreate |
CFile::typeBinary);
                //写位图文件头
                bitmapFile.Write(&hdr, sizeof(BITMAPFILEHEADER));
                //写位图数据
                bitmapFile.Write(buffer, bitmapSize);
                bitmapFile.Close();   //关闭位图文件
                pass = true;          //抓图成功
              }
              delete [] buffer;       //释放缓冲区
              return pass;
       }
     }
     return false;
 }
```

上述程序中首先获取图像的位图格式的大小，然后获取图像数据，接着创建 BMP 位图

的头信息等，最后写文件头和实际的图像数据。

8.3 实现一个简单的媒体播放器

最简单的媒体播放器包括打开、播放和停止文件。现在我们就来实现这个最简单的播放器。首先根据前面介绍的 VC++ 2005 创建对话框应用程序的步骤，创建一个最基本的程序框架，然后添加 3 个按钮和 1 个图像控制控件，如图 8-28 所示。详细步骤见 8.1.3 小节的“界面设计”，这里只介绍按钮事件响应过程。

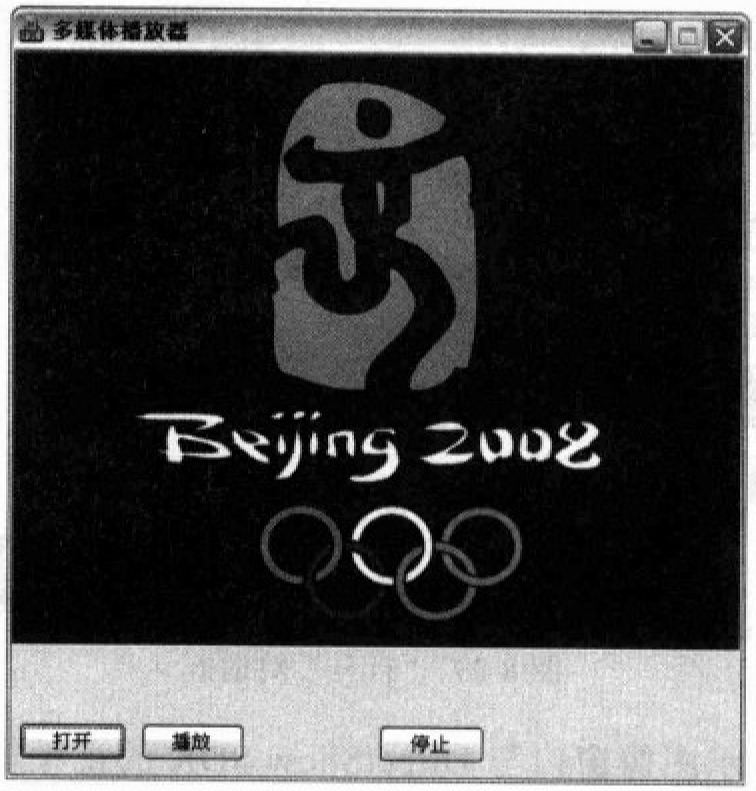


图 8-28　一个简单的媒体播放器

为“打开”按钮添加鼠标单击事件响应，双击按钮，代码如下。

```
void CMediaPlayerDlg::OnBnClickedButtonOpen()
{
 //TODO: 在此添加控件通知处理程序代码
 //文件类型过滤表
 CString strFilter = _T("AVI File (*.avi) | *.avi|");
 strFilter += "MPEG File (*.mpg; *.mpeg) | *.mpg; *.mpeg|";
        strFilter += "MP3 File (*.mp3) | *.mp3|";
        strFilter += "WMA File (*.wma) | *.wma|";
        strFilter += "All File (*.*) | *.*|";
        //创建打开文件对话框对象
        CFileDialog dlg(TRUE,
                        NULL, NULL,
                        OFN_PATHMUSTEXIST|OFN_HIDEREADONLY,
                        strFilter, this);
        if (dlg.DoModal() == IDOK)
        {
                //获取媒体路径和文件名
```

```
                m_sourceFile = dlg.GetPathName();
                 //从路径中获取多媒体文件名称
                m_mediaFileName =GetFileTitleFromFileName(m_sourceFile,1);
                CreateGraph();//创建链表，连接滤波器
        }
}
```

在上述程序中，以只读方式打开文件，且过滤出流媒体文件格式。获取媒体的路径和文件名。然后根据媒体文件名创建滤波器链表，链接这些滤波器。剖析媒体文件格式，自动查询对应的解码器，并播放第一帧流媒体数据，然后暂停，上述程序的运行效果如图 8-29 所示。

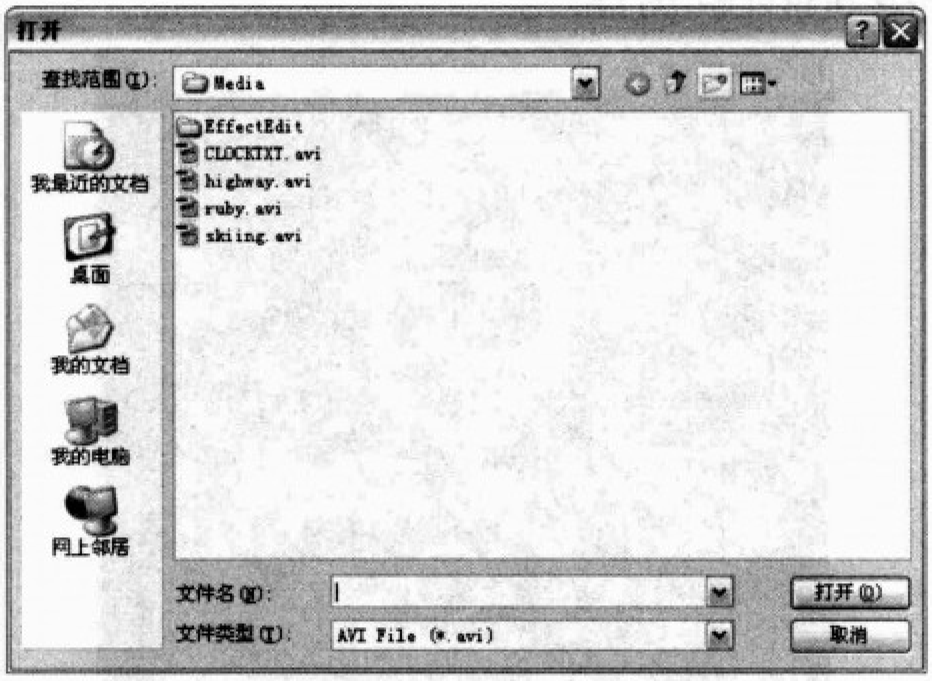


图 8-29 “打开”对话框

渲染媒体文件，把显示图像窗口与 DirectShow SDK 的视频窗口接口捆绑，代码如下。

```
void CMediaPlayerDlg::CreateGraph()
{
 DestroyGraph();                          //销毁滤波器链表图
 m_pFilterGraph = new CDXGraph();         //创建CDXGraph对象
 if (m_pFilterGraph->Create())            //创建滤波器链表管理器
 {
       //UNICODE模式下，把CString转换为Char
       CString str=this->m_sourceFile;
       wchar_t *wchar;
       wchar = str.GetBuffer();
       str.ReleaseBuffer();
       size_t convertedChars = 0;
       size_t  sizeInBytes = ((str.GetLength() + 1) * 2);
       char *ch = (char *)malloc(sizeInBytes);
       wcstombs_s(&convertedChars, ch, sizeInBytes, wchar, sizeInBytes);
       if (!m_pFilterGraph->RenderFile(ch)) //渲染媒体文件，构建滤波器链表
       {
           MessageBox(_T("无法渲染此媒体文件！请确认是否安装相关解码器插件！\n 或者此媒
体文件已损坏！"),_T("系统提示"),MB_ICONWARNING);
           return;
```

```
        }
            //设置图像显示窗口
        m_pFilterGraph->SetDisplayWindow(m_videoWindow.GetSafeHwnd());
          //设置窗口消息通知
        m_pFilterGraph->SetNotifyWindow(this->GetSafeHwnd());
        //显示第一帧图像
        m_pFilterGraph->Pause();
 }
}
```

由于所有对 DirectShow SDK 的控制是封装在类 CDXGraph 中实现的，所以首先创建一个 CDXGraph 对象，然后创建滤波器链表管理器，接着把读取的文件路径名修改为宽字符形式。渲染媒体文件，自动剖析媒体格式，构建滤波器链表。若渲染成功，则设置图像显示窗口并注册消息通知窗口。最后显示第一帧图像后，马上暂停，效果如图 8-30 所示。

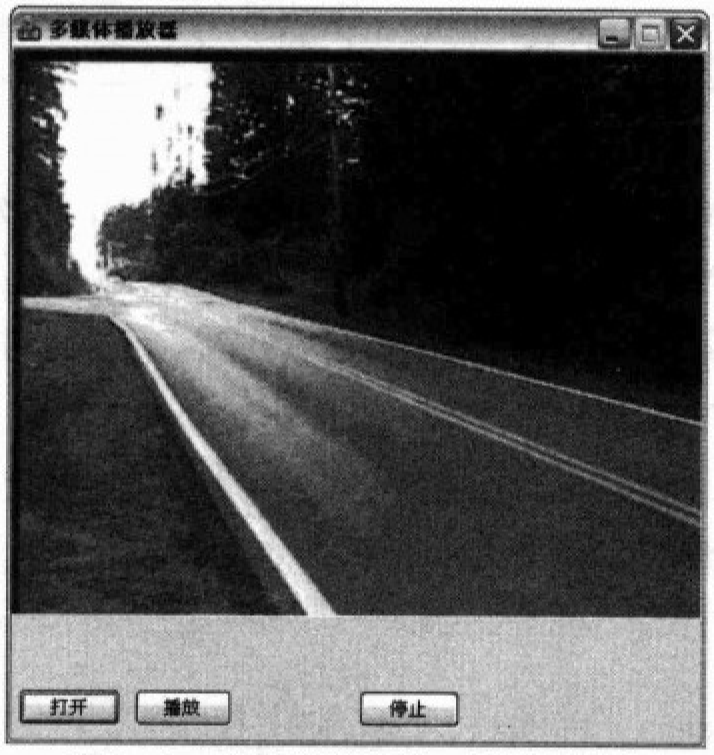


图 8-30 打开并播放第一帧媒体文件

开始播放媒体文件，同时把播放速度和媒体文件名在程序的标题栏显示，代码如下。

```
void CMediaPlayerDlg::OnBnClickedButtonPlay()
{
 //TODO: 在此添加控件通知处理程序代码
 if (m_pFilterGraph)          //CDXGraph对象不为空
 {
      //把媒体文件名设置到对话框标题
      SetWindowText(_T("1倍速播放") + m_mediaFileName);
      m_pFilterGraph->Run();//开始运行媒体
 }
}
```

上述代码在 CDXGraph 对象不为空的情况下，首先设置媒体文件名到对话框的标题栏，然后开始运行媒体。假定在图 8-29“打开”对话框中选择 D:\dxsdk\samples\media 下的 highway.avi 文件，然后单击“运行”按钮，则上述代码的效果如图 8-31 所示。

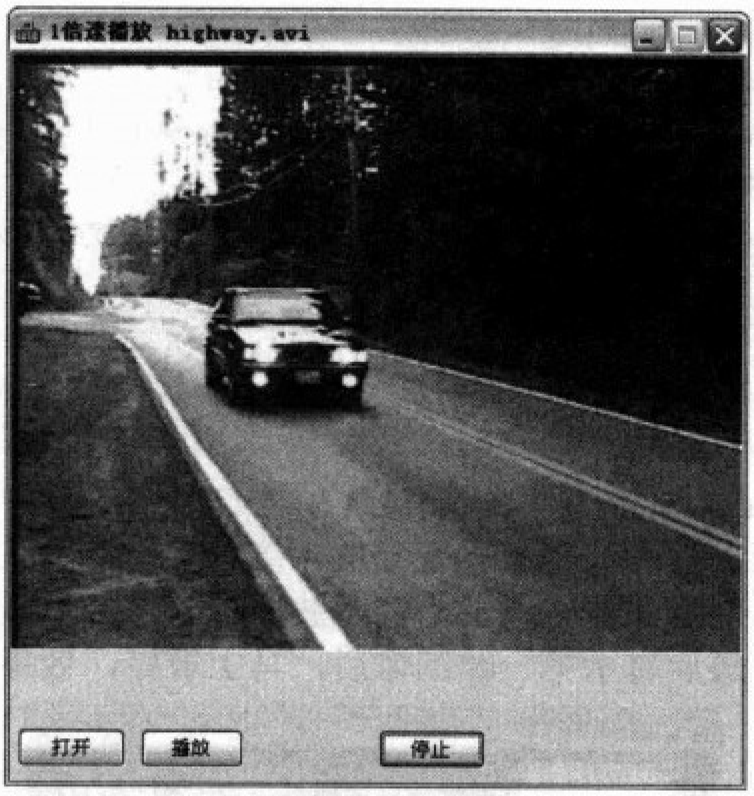


图 8-31　运行媒体效果图

为了捕获媒体播放的消息和各种事件，向窗口发送通知，步骤如下。

STEP 01 向对话框的类中添加自定义的消息处理函数 OnGraphNotify。

```
afx_msg LRESULT OnGraphNotify(WPARAM inWParam, LPARAM inLParam);
```

STEP 02 向对话框的消息映射部分添加消息映射宏。

```
ON_MESSAGE(WM_GRAPHNOTIFY, OnGraphNotify)
```

STEP 03 消息处理函数的实现，代码如下。

```
LRESULT CMediaPlayerDlg::OnGraphNotify(WPARAM inWParam, LPARAM inLParam)
{
 IMediaEventEx *pEvent = NULL;    //媒体事件接口指针
 //获取事件句柄
 if ((m_pFilterGraph!=NULL) && (pEvent = m_pFilterGraph->GetEventHandle()))
 {
       LONG eventCode = 0;          //事件码
       LONG eventParam1 = 0;        //事件码的第一个参数
       LONG eventParam2 = 0;        //事件码的第二个参数
 //获取事件码及其两个参数
while (SUCCEEDED(pEvent->GetEvent(&eventCode, &eventParam1, &eventParam2, 0)))
       {
           //获取成功，则释放事件及其参数
           pEvent->FreeEventParams(eventCode, eventParam1, eventParam2);
           switch (eventCode)       //根据事件码内容，分别处理各种消息
           {
           case EC_COMPLETE:        //播放结束
                OnBnClickedButtonPause();                   //暂停播放
                m_pFilterGraph->SetCurrentPosition(0); //设置到媒体开始位置
                break;
           case EC_USERABORT:       //用户终止消息
```

```
                case EC_ERRORABORT:     //出错终止消息
                    OnBnClickedButtonStop();
                    break;
                default:
                    break;
                }
            }
        }
        return 0;
}
```

上述程序实现消息处理，首先获取事件及其参数，然后根据事件码的内容分别进行处理。注意获取事件后，需马上释放事件及其参数。

8.4 升级你的播放器

前面我们已经实现了一个简单的播放器，现在增强播放器的功能，如菜单控制（媒体播放控制、播放速率控制、静音开关、全屏开关、置顶显示、抓图等）、音量调节、媒体拖放定位等。

8.4.1 添加控制功能

播放器主界面上列出了最常用的控制，而菜单实现了播放器的所有功能。下面给出了主要的控制实现，其他详见光盘。

1）*视频显示窗口的右键快捷菜单*

为了在视频中显示窗口，单击鼠标右键激活菜单，需要我们重载窗口消息捕获处理函数 PreTranslateMessage。

STEP 01 选择 VC++ 2005 开发环境中的“视图”→“类视图”命令打开“类视图”属性页。选中 CMediaPlayerDlg 类，单击鼠标右键，在弹出的快捷菜单中选择“属性”命令，打开“属性”对话框，如图 8-32 所示。单击“重写”按钮，查询定位 PreTranslateMessage 函数，然后单击“添加”按钮。

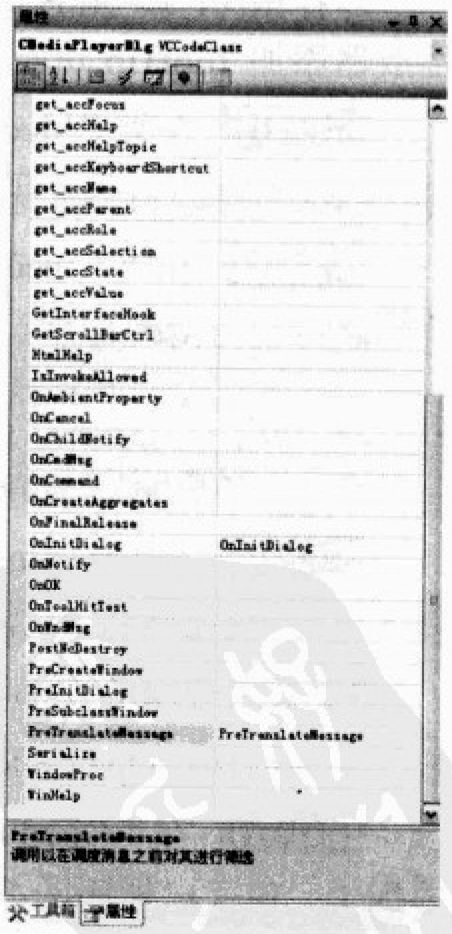


图 8-32 “属性”对话框

STEP 02 虚拟函数的实现，代码如下。

```
BOOL CMediaPlayerDlg::PreTranslateMessage(MSG* pMsg)
{
 {
        //让tooltip处理如下消息
        m_tooltip.RelayEvent(pMsg);
```

```
    }
        //命令是按下鼠标右键
    if (WM_RBUTTONDOWN == pMsg->message)
    {
        CPoint point;
        CMenu hmenu;
        HMENU hmenuTrackPopup;
        hmenu.LoadMenu(IDR_MENU_MAIN); //装载菜单
        GetCursorPos(&point);          //获取鼠标当前位置
        hmenuTrackPopup = GetSubMenu(hmenu, 0);
        //弹出式菜单
        TrackPopupMenu(hmenuTrackPopup, TPM_LEFTALIGN | TPM_LEFTBUTTON,
                point.x, point.y, 0, this->m_hWnd, NULL);
        DestroyMenu(hmenu);
    }
    return CDialog::PreTranslateMessage(pMsg);
}
```

上述程序实现对按下鼠标右键的命令进行处理，按下时，获取当前光标位置，然后装载菜单，再获取其子项，最后显示菜单，别忘记在最后要销毁该菜单。

2）按钮在线提示 ToolTip

在使用 VC++ 2005 时，鼠标停留某个菜单或工具按钮片刻会有在线提示。现在就给媒体播放器添加 ToolTip 功能，具体步骤如下。

STEP 01 在类 CMediaPlayerDlg 定义中声明 tooltip 控件。

```
CToolTipCtrl m_tooltip;
```

STEP 02 在类 CmediaPlayerDlg 实现文件的对话框初始化函数 OnInitDialog 中添加：

```
m_tooltip.Create(this);
 m_tooltip.Activate(TRUE);
 m_tooltip.AddTool(GetDlgItem(IDC_BUTTON_OPEN), _T("Open Media File"));
    //添加其他按钮的tooltip
```

STEP 03 在 PreTranslateMessage 消息处理函数中添加：

```
m_tooltip.RelayEvent(pMsg);
```

3）菜单子项响应函数

下面给出“Open File”菜单项响应函数的过程。

STEP 01 在“Open File”菜单上单击鼠标右键，在弹出的快捷菜单中选择“添加事件处理程序”命令，如图 8-33 所示，其他菜单子项响应函数完全与此类似。

STEP 02 选择消息类型“COMMAND”，在类列表中选中“CMediaPlayerDlg”选项，单击“添加编辑”按钮。

```
void CMediaPlayerDlg::OnMenuOpenfile()
{
 //TODO: 在此添加命令处理程序代码
```

```
 OnBnClickedButtonOpen();
}
```

直接调用按钮的消息处理函数。

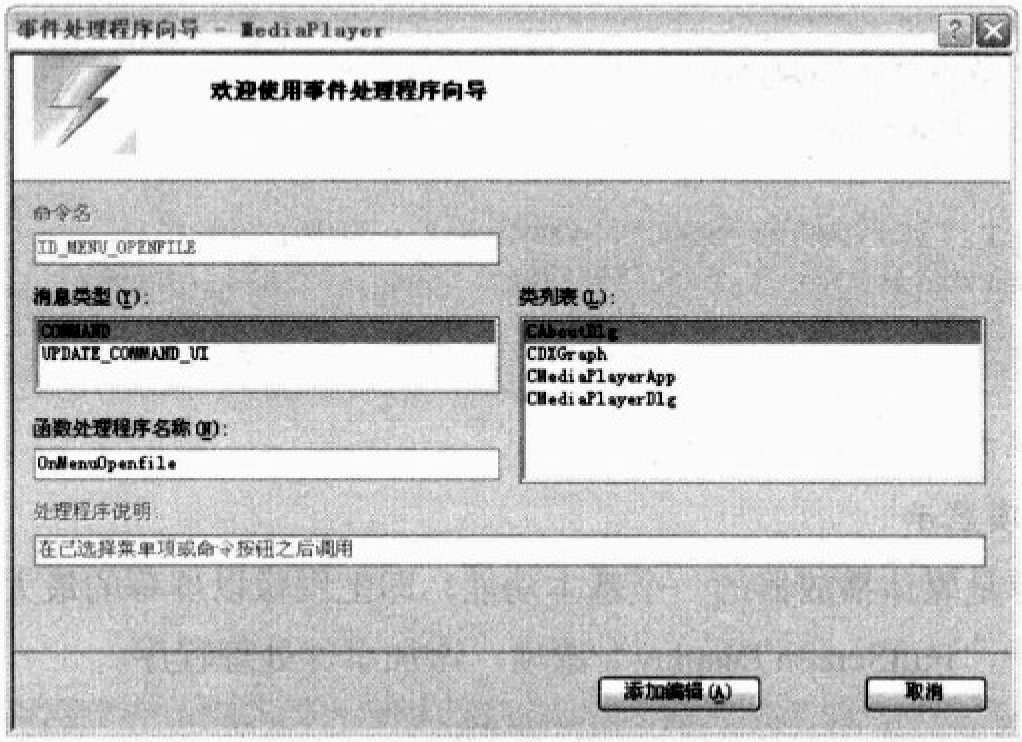


图 8-33　事件处理程序向导

4）播放速率控制

媒体播放可以是慢放、快放和正常播放，下面是功能实现代码。需要提及的是，对于ASF、WMV 这样一些 DRM 媒体格式，在这里不支持播放速率控制。

STEP 01 选中“Half Rate”选项，以 1/2 速率回放媒体，添加事件处理程序。

```
void CMediaPlayerDlg::OnMenuHalfrate()
{
 //TODO: 在此添加命令处理程序代码
 if (m_pFilterGraph)
 {
       m_pFilterGraph->SetPlaybackRate(0.5); //设置回放速率为0.5
       SetWindowText(_T("1/2倍速播放") + m_mediaFileName);//设置标题栏
 }
}
```

设置回放速率小于 1 表示慢放。

STEP 02 选中“Normal Rate”选项，以单倍速率回放媒体，添加事件处理程序。

```
void CMediaPlayerDlg::OnMenuNormalrate()
{
 //TODO: 在此添加命令处理程序代码
 if (m_pFilterGraph)
 {
       m_pFilterGraph->SetPlaybackRate(1.0);
       SetWindowText(_T("1倍速播放") + m_mediaFileName);
 }
}
```

设置回放速率等于 1，表示正常速率播放。

STEP 03 选中“Double Rate”选项，以 2 倍速率回放媒体，添加事件处理程序。

```
void CMediaPlayerDlg::OnMenuDoublerate()
{
    //TODO: 在此添加命令处理程序代码
    if (m_pFilterGraph)
    {
        m_pFilterGraph->SetPlaybackRate(2.0);
        SetWindowText(_T("2倍速播放") + m_mediaFileName);
    }
}
```

设置回放速率大于 2，表示要快速回放。

5）全屏和置顶显示

全屏显示图像是媒体播放器的一个基本功能，即把图像以屏幕的最大规格显示图像。

STEP 01 选中“FullScreen Display”选项，添加事件处理程序。

```
void CMediaPlayerDlg::OnMenuFullscreen()
{
 //TODO: 在此添加命令处理程序代码
 static int flag=0;
 if (m_pFilterGraph != NULL)
 {
        if (!flag){
            m_pFilterGraph->SetFullScreen(TRUE);
            flag = 1;
        }else{
            m_pFilterGraph->SetFullScreen(FALSE);
            flag = 0;
        }
    }
}
```

上述程序实现全屏显示开关，使用了局部 static 变量记录信息。

STEP 02 实现退出全屏显示，程序捕获“Esc”键消息，在 PreTranslateMessage 中添加消息捕获处理。

```
if (pMsg->message == WM_KEYDOWN)
 {
        //按键盘的回车键或“Esc”键
        if (pMsg->wParam == VK_RETURN || VK_ESCAPE)
        {
            if (m_pFilterGraph != NULL)
            {
                    if (m_pFilterGraph->GetFullScreen())
                                                //获取当前是否是全屏显示
```

```
                {
                        m_pFilterGraph->SetFullScreen(FALSE); //关闭全屏显示
                }
            }
            return TRUE;
        }
    }
```

按回车键或“Esc”键均可以从全屏显示中恢复出来。首先获取当前显示是否为全屏状态，若是则关闭全屏显示。

STEP 03 视频窗口置顶显示，从而使图像窗口我们始终可见，添加事件处理程序。

```
void CMediaPlayerDlg::OnMenuAlwaysontop()
{
//TODO: 在此添加命令处理程序代码
static int flag=0;
if (!flag){
//置顶显示HWND_TOPMOST
::SetWindowPos(m_hWnd,HWND_TOPMOST,0,0,0,0,SWP_NOMOVE | SWP_NOSIZE);
flag = 1;
 }else {
 //不置顶显示HWND_NOTOPMOST
   ::SetWindowPos(m_hWnd,HWND_NOTOPMOST,0,0,0,0,SWP_NOMOVE | SWP_NOSIZE);
flag = 0;
}
}
```

视频窗口的置顶显示是为了在其他窗口激活时，能够继续浏览视频。专业的播放器都有该功能。

8.4.2 添加拖放功能

媒体文件在播放时，可以任意拖动滑动条以实现媒体文件的定位播放，同时滑动条定时滚动对应媒体播放。

STEP 01 滑动条控件与变量捆绑，在滑动条上单击右键，“添加变量”名为“m_sliderPlayer”。

STEP 02 滑动条变量初始化，在 OnInitDialog 函数中添加：

```
m_sliderPlayer.SetRange(0, 1000);     //媒体播放滑动条范围
m_sliderPlayer.SetPos(0);             //设置当前位置为0
```

STEP 03 添加水平滚动消息处理函数。选中播放器主对话框，单击鼠标右键，在弹出的快捷菜单中选择“属性”命令，在打开的对话框中单击“消息”按钮 。查找 WM_HSCROLL，添加 OnHScroll 消息处理函数。

```
void CMediaPlayerDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{//TODO: 在此添加消息处理程序代码和/或调用默认值
 if (pScrollBar->GetSafeHwnd() == m_sliderPlayer.GetSafeHwnd())
```

```
    {
        if (m_pFilterGraph != NULL)
        {
            double duration =1.0;
            m_pFilterGraph->GetDuration(&duration); //获取媒体文件时间长度
            //获取滑动条当前位置
            double pos=duration*m_sliderPlayer.GetPos()/1000.0;
            m_pFilterGraph->SetCurrentPosition(pos);//设置当前位置
        }
    } else
    {
        CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
    }
}
```

上述程序实现对播放的滑动条的事件的处理，首先获取媒体文件的时间长度，然后获取滑动条的位置，最后设置当前位置，效果如图 8-34 所示。

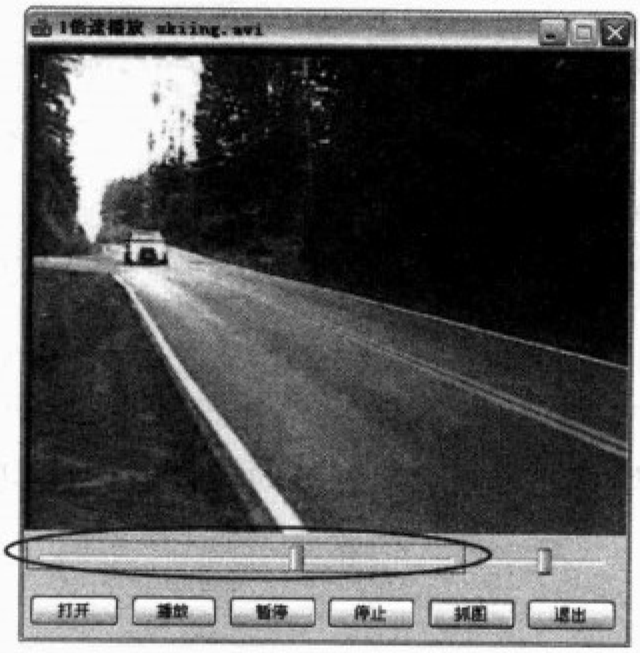


图 8-34　播放滑动条控制

8.4.3　添加音量调节功能

音量调节类似于媒体定位，使用滑动条实现调整。

STEP 01　滑动条与变量捆绑，定义变量 m_sliderVolume。

STEP 02　初始化滑动条。

```
m_sliderVolume.SetRange(50,100);
m_sliderVolume.SetPos(50);
```

STEP 03　添加水平滚动条消息处理函数。

```
void CMediaPlayerDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
 //TODO: 在此添加消息处理程序代码和/或调用默认值
```

```
if (pScrollBar->GetSafeHwnd() == m_sliderPlayer.GetSafeHwnd())
{ //滑动条为播放进度
      if (m_pFilterGraph != NULL)
      {
            double duration =1.0;
            m_pFilterGraph->GetDuration(&duration);  //获取文件长度
            //获取当前回放的位置
            double pos = duration * m_sliderPlayer.GetPos()/1000.0;
            m_pFilterGraph->SetCurrentPosition(pos); //重新设置当前位置
      }
}else if (pScrollBar->GetSafeHwnd() == m_sliderVolume.GetSafeHwnd())
{ //滑动条为音量控制
      if (m_pFilterGraph != NULL)
      {
            m_volume = m_sliderVolume.GetPos(); //获取当前音频控制位置
            m_pFilterGraph->ChangeAudioVolume(m_volume);//更改音频大小
      }
} else {
      CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
}
}
```

该函数是所有滑动条的消息处理函数，根据窗口句柄识别具体的滑动条，然后获取当前滑动条的位置，最后设置当前参数，图 8-35 所示为音量控制界面。

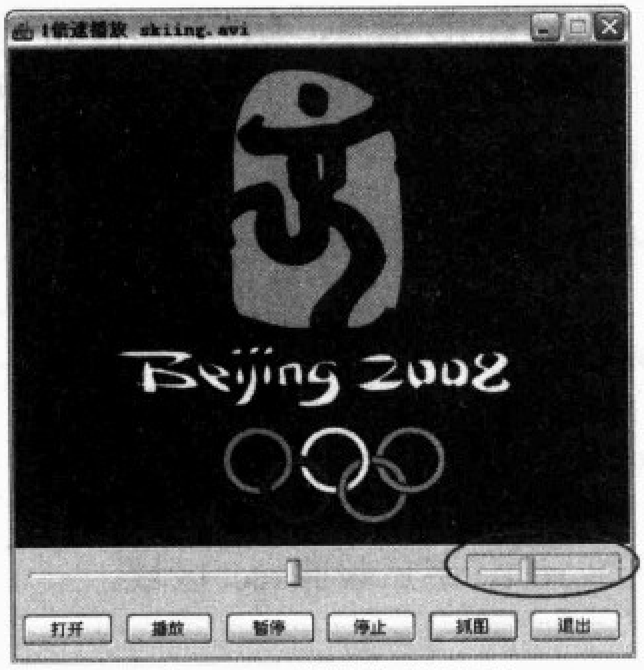


图 8-35 音量控制界面

STEP 04 静音控制，菜单响应实现代码。

```
void CMediaPlayerDlg::OnMenuMute()
{
 //TODO: 在此添加命令处理程序代码
 if (m_pFilterGraph != NULL)        //CDXGraph对象不为空
 {
      static int flag=0;            //静音标识
```

```
        if (!flag){                         //没有静音
              m_pFilterGraph->Mute();      //设置为静音
              flag = 1;
        }else{                              //已经静音
              m_pFilterGraph->UnMute(); //关闭静音
              flag = 0;
        }
  }
}
```

上述程序的静音控制是开关，使用了 static 类型，即程序局部修改单值会全局记忆。

8.4.4 添加欢迎背景图片

一方面我们实现程序欢迎界面，另一方面当媒体文件没有视频时，播放的只有音频数据。现在我们实现在图像显示窗口中显示特定背景。

STEP 01 加载背景图片资源，格式可以是 BMP、JPEG、GIF 等。

STEP 02 修改图像控件 IDC_VIDEO_WINDOW 的属性，设置“Type”为“Bitmap”，“Sunken”为“True”，“Image”为“IDB_BITMAP_BKGROUND”。

STEP 03 设置图像显示窗口 m_videoWindow 的模式，在主对话框的初始化中，添加：

```
m_videoWindow.ModifyStyle(0, WS_CLIPCHILDREN);
```

STEP 04 重载 WM_ERASEBKGND 消息，为图像控件 m_videoWindow 创建一个新的剪切区域，以保证其他窗口覆盖图像显示窗口后，再正常显示以前的图像。

```
BOOL CMediaPlayerDlg::OnEraseBkgnd(CDC* pDC)
{
 //TODO: 在此添加消息处理程序代码和/或调用默认值
 CRect rect;                             //矩形类对象
 m_videoWindow.GetClientRect(&rect); //获取客户区域
 pDC-> ExcludeClipRect(&rect);          //裁剪客户区域
 return CDialog::OnEraseBkgnd(pDC);    //绘制背景
}
```

上述程序首先获取客户区域 rect，然后裁剪 rect 区域。

8.5 系统效果展示

前面展示了 DirectShow SDK 实现流媒体回放的工作原理和实现步骤，现在就使用设计完成的应用程序 MediaPlayer 播放音频、视频媒体文件，感受系统的效果。这里包括纯音频、音/视频组合的媒体文件。

这里播放 MP3 文件。首先运行媒体播放器，选中要播放的文件，然后“打开”文件，如图 8-36 所示，最后“播放”媒体文件，如图 8-37 所示。

图 8-36 打开音频文件

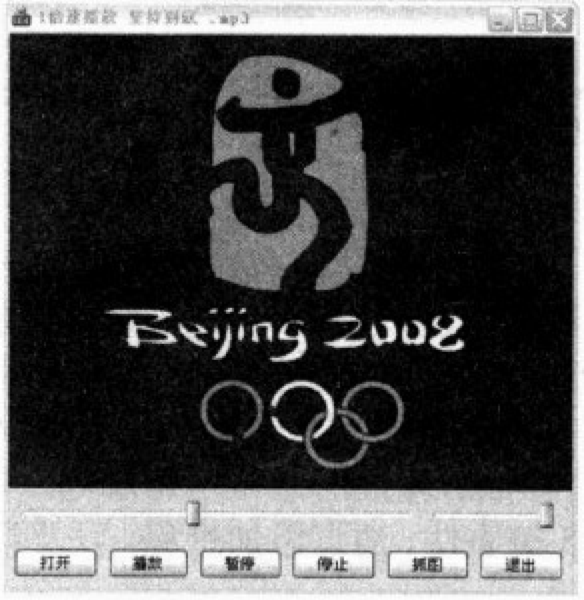


图 8-37 播放音频

如播放*.avi 文件。首先运行媒体播放器，选择要播放的 AVI 文件，然后单击“打开”按钮，最后“播放”媒体文件，如图 8-38 所示。

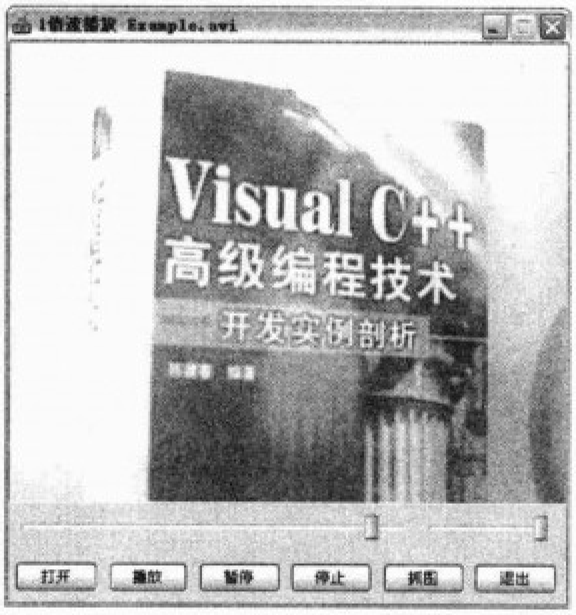


图 8-38 播放音/视频

8.6 如何播放更多的文件格式

在使用媒体播放器时，某些下载的电影或媒体文件无法播放，且出现图 8-39 所示的画面，一方面说明系统没有对应的解码器，另一方面文件可能不是一个多媒体文件或媒体格式不标准。想播放更多的媒体，就需要清楚媒体的格式，或者安装“万能播放器”插件，就可以播放各种格式的媒体文件了。

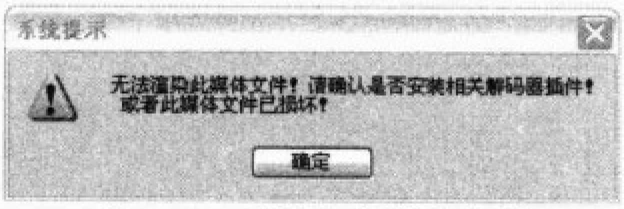


图 8-39 无法渲染媒体文件

8.7 本章小结

本章以多媒体播放器案例为设计内容，剖析了使用 DirectShow SDK 开发流媒体播放器的原理和设计过程，封装了媒体播放类 CDXGraph 完成媒体控制的几乎所有的功能，该类包括创建滤波器链表管理器、查询各种接口，滤波器链表自动剖析媒体格式，选用合适的滤波器。对媒体的控制包括打开媒体文件、播放/暂停/停止媒体、全屏显示、音量控制、抓图、静音控制、播放时总位于顶层等。使用 VC++ 2005 设计友好的主界面和菜单，展示了 VC++ 2005 的事件、消息添加过程，以及常用控件 CButton、Slider Ctronl 等的设计和使用技术要点。

第 9 章

自制 DV 实现视频采集

视频采集是 DirectShow 除了媒体回放以外的又一基本功能。滤波器（Filters）、COM 接口屏蔽了我们对硬件的访问和复杂的采集过程，使用 DirectShow SDK 快速构建视频捕获应用程序，实现视频采集或视频图像捕获。我们采集视频的目的，一方面直接存盘成为一个视频文件；另一方面期望以图像帧的方式，采集 RGB/YUV 格式数据，对数据做直接处理：滤波、编码、识别等。

本章技术要点是基于 DirectShow SDK 实现视频、图像的预览与采集，从功能实现类的分析到案例界面设计，详细介绍视频采集的有关知识。以“使用经典采集技术实现视频捕获”实例分析视频流捕获、存盘过程。即使用 DirectShow SDK 的 ICaptureGraphBuilder2 标准接口，利用该接口的所属方法 RenderStream 自动建立、链接滤波器链表，该过程对用户来说视频数据是透明的，无法直接访问。以“使用 VMR 技术实现视频图像捕获”实例来分析图像流畅显示和图像帧捕获过程。即使用 VMR（Video Mixer Render）在流畅预览视频的同时，捕捉图像帧，然后从颜色空间 RGB 转换到 YUV420/I420，供视频图像处理算法直接处理，或直接保存为 RGB/YUV 格式的文件。本案例就分别以两个实例介绍 DirectShow SDK 的视频采集、预览和存盘的技术过程。

本章重点

- 系统分析与设计
- 使用经典采集技术实现视频捕获
- 使用 VMR 技术实现视频图像捕获
- 系统效果演示

9.1 系统分析与设计

视频数据的实时采集、捕获是视频处理的前提。随着计算机 CPU 性能的迅猛提升，视频数据的计算机算法实时软处理，对如 DSP、ASIC 芯片硬件算法处理方式形成一定的威胁。并且，出现了许多与计算机直接连接的设备：数码摄像机 DV、数字照相机 DV、USB 摄像头等。我们可以对这些设备直接访问，轻松获取视频数据。

9.1.1 采集设备的枚举

使用采集设备前，需要首先确定系统已经安装的采集设备：视频、音频采集设备。系统设备枚举器为按类型枚举已注册在系统中的滤波器提供了统一的方法。而且它能够区分不同的硬件设备，即便是同一个滤波器支持它们。这对那些使用 Windows 驱动模型、KSProxy Filter 的设备来说是非常有用的，系统设备枚举器对它们按不同的设备实例进行对待。

当利用系统设备枚举器查询设备的时候，系统设备枚举器为特定类型的设备（如音频捕获和视频压缩）生成了一张枚举表（Enumerator）。类型枚举器（Category Enumerator）为每个这种类型的设备返回一个 Moniker，类型枚举器自动把每种即插即用的设备包含在内。

调用标准方法 CoCreateInstance 生成系统设备枚举器（Device Enumerator），类标识（CLSID）为 CLSID_SystemDeviceEnum，代码如下。

```
//枚举捕获设备
 ICreateDevEnum *pCreateDevEnum;  //创建设备枚举器
//创建设备枚举管理器
 HRESULT hr = CoCreateInstance(CLSID_SystemDeviceEnum,//要创建的Filter的Class ID
                              NULL,                    //表示Filter不被聚合
                              CLSCTX_INPROC_SERVER,    //创建进程内COM对象
                              IID_ICreateDevEnum,      //获得的接口ID
                              (void**)&pCreateDevEnum); //创建的接口对象的指针
 if (hr != NOERROR)           return -1;
```

函数原型为：

```
        HRESULT CoCreateInstance(const CLSID& clsid,
                              IUnknown *pUnknownOuter,
                              DWORD dwClsContext,
                              const IID& iid,
                              (void **)ppv);
```

第 1 个参数 clsid 是一个 GUID，它可唯一指定客户端需要使用的 COM 类，GUID 或者 CLSID 是 COM 类的标识符，世界上的每个 COM 类都有自己唯一的 CLSID。COM 将使用该 ID 来查找可产生请求 COM 对象的服务器，一旦连接到服务器，即会创建该对象。

第 2 个参数 pUnknownOuter 是一个指针，它指向“outer unknown”。我们不会使用这个

参数，因此传送NULL。在涉及“aggregation”（聚合）概念时，outer unknown是很重要的，aggregation 可让一个接口直接调用另一个 COM 接口而无须通知客户端，aggregation 和 containment是接口用来调用其他接口的两个方法。

第3个参数dwClsContext定义COM类的Context或者CLSCTX。该参数控制服务器的范围，我们可以通过它来控制服务器是进程内的服务器还是一个EXE或者是在远程的计算机上。CLSCTX 是一个位掩码，因此用户可以混合几个值，这里我们使用的是 CLSCTX_INPROC_SERVER，该服务器将运行在本地的计算机（进程内）上，并且作为一个DLL连接到客户。

第4个参数iid是接口的标识符或者IID。这是另一个GUID，用来标识我们请求的接口。我们请求的IID必须是存在的，即被由CLSID指定的COM类支持，再次，IID的值通常由一个头文件提供，或者使用接口名查找出来。

第5个参数ppv是指向一个接口的指针。CoCreateInstance() 将创建所请求的类对象和接口，并且返回一个指向接口的指针，这个参数也是 CoCreateInstance 调用的目的，然后我们就可以使用该接口指针来调用服务器上的方法。

调用 ICreateDevEnum::CreateClassEnumerator 方法生成类型枚举器，参数为用户想要得到的类的ID（CLSID），该方法返回一个IEnumMoniker接口指针。如果指定的类型是空的或不存在，则函数ICreateDevEnum::CreateClassEnumerator将返回S_FALSE而不是错误代码，同时 IEnumMoniker 指针也是空的，这就要求我们在调用 CreateClassEnumerator 的时候明确用S_OK进行比较而不使用宏SUCCEEDED，代码如下。

```
IEnumMoniker *pEm;                  //枚举监控器接口
 //获取视频类的枚举器
 hr=pCreateDevEnum->CreateClassEnumerator(CLSID_VideoInputDeviceCategory,
     &pEm, 0);
 //如果想获取音频类的枚举器，则使用如下代码
 //hr=pCreateDevEnum->CreateClassEnumerator(CLSID_AudioInputDeviceCategory,
   &pEm, 0);
 if (hr != NOERROR) return -1;
```

函数原型如下：

```
HRESULT CreateClassEnumerator(
  REFCLSID clsidDeviceClass,
  IEnumMoniker **ppEnumMoniker,
  DWORD dwFlags
);
```

clsidDeviceClass 表示设备种类的类标识符 CLSID；ppEnumMoniker 是接收 IEnumMoniker接口类型的指针；dwFlags表示枚举的种类标识，为0表示枚举所有类型。

类型枚举器复位后，使用IEnumMoniker::Next方法依次得到IEnumMoniker指针中的每个Moniker，该方法返回一个IMoniker接口指针。当Next到达枚举的底部时，它的返回值仍然是S_FALSE，这里仍需要用S_OK来进行检验。要想得到该设备较为友好的名称（如想要用户界面在组合框中显示），则调用IMoniker::BindToStorage方法，代码如下。

```
pEm->Reset();                                          //类型枚举器复位
ULONG cFetched;
 IMoniker *pM;                                         //监控器接口指针
 while(hr = pEm->Next(1, &pM, &cFetched), hr==S_OK)    //获取下一个设备
 {
       IPropertyBag *pBag;                             //属性页接口指针
       hr = pM->BindToStorage(0, 0, IID_IPropertyBag, (void **)&pBag);
                                                       //获取设备属性页
       if(SUCCEEDED(hr)) {
             VARIANT var;
             var.vt = VT_BSTR;                            //保存的是二进制数据
             hr = pBag->Read(L"FriendlyName", &var, NULL);
                                                   //获取FriendlyName形式的信息
             if (hr == NOERROR)
             {
                   id++;
                   // 把FriendlyName信息添加到组合列表框
                   ::SendMessage(hList, CB_ADDSTRING, 0,(LPARAM)var.bstrVal);
                   SysFreeString(var.bstrVal);     //释放资源，特别要注意
             }
             pBag->Release();                      //释放属性页接口指针
       }
       pM->Release();                              //释放监控器接口指针
 }
```

函数原型如下：

```
HRESULT Next(
        ULONG celt,
        IMoniker **rgelt,
        ULONG *pceltFetched)
```

celt 表示要查询的对象的数目；rgelt 表示返回的接口对象的指针；pceltFetched 是返回的包含 rgelt 对象的实际数目。函数原型如下：

```
HRESULT BindToStorage(
        IBindCtx *pbc,
        IMoniker *pmkToLeft,
        REFIID riid,
        void **ppvObj)
```

pbc 参数指向捆绑环境接口 IbindCtx；pmkToLeft 指向监控器的左边监控器，一般该值设置为空；iid 参数是监控器的类标识符；ppvObj 指向获取接口的指针。

以上的 3 个步骤封装在函数 EnumDevices 中实现，参数为显示设备的窗口句柄。代码如下。

```
int CCaptureClass::EnumDevices(HWND hList)
{
```

```
    if (!hList) return -1;                    //句柄为空则返回一1
    int id = 0;
  //上述的3个步骤
    return id;
}
```

该函数的入口参数为hList，它是应用程序传入的组合框的句柄。在查询到视频捕获设备后，把设备的名字添加到组合框中。

9.1.2 使用Capture Graph Builder

使用DirectShow的滤波器进行流媒体开发，需要一个管理器即滤波器链表管理器（Filter Graph Manager），DirectShow SDK提供的是Graph Builder接口IgraphBuilder。不过针对捕获任务（Capture），还有另一个接口ICaptureGraphBuilder2针对采集捕获的增强型接口。

```
/*创建滤波器链表管理器，查询其各种控制接口*/
HRESULT CCaptureClass::InitCaptureGraphBuilder()
{
    HRESULT hr;
  //创建IGraphBuilder接口
  hr = CoCreateInstance(CLSID_FilterGraph, NULL,
                          CLSCTX_INPROC_SERVER,
                          IID_IGraphBuilder, (void **)&m_pGB);
  if (FAILED(hr)) return hr;
  //创建ICaptureGraphBuilder2接口
  hr = CoCreateInstance(CLSID_CaptureGraphBuilder2 , NULL,
                          CLSCTX_INPROC,
                          IID_ICaptureGraphBuilder2, (void **)&m_pCapture);
  if (FAILED(hr)) return hr;
  //初始化滤波器链表管理器IGraphBuilder
  m_pCapture->SetFiltergraph(m_pGB);
  //查询媒体控制接口
  hr = m_pGB->QueryInterface(IID_IMediaControl, (void **)&m_pMC);
  if (FAILED(hr)) return hr;
  //查询视频窗口接口
  hr = m_pGB->QueryInterface(IID_IVideoWindow, (LPVOID *) &m_pVW);
  if (FAILED(hr)) return hr;
  return hr;
}
```

在上述程序中，首先创建IGraphBuilder接口，然后创建ICaptureGraphBuilder2接口，接着使用ICaptureGraphBuilder2初始化IGraphBuilder接口。在IGraphBuilder接口下查询媒体控制接口、视频窗口接口。

9.1.3 采集参数的设置

采集前需要对要采集的视频格式、图像质量进行设置，如视频的分辨率、帧率和数据格式，图像的亮度、色度和饱和度参数设置等。下面分别对这两种设置进行讲述。

1. 视频格式设置

对采集设备的参数进行配置，首先确定设备的类型：**VFW**、**WDM** 驱动模型。目前流行的采集设备大部分采用 WDM 模式，采集设备在 DirectShow 中被当做一个滤波器组件，所以配置采集设备就是对该滤波器的引脚进行设置。在确保滤波器链表完全建立前，可以使用如下的过程配置视频格式。

```
/*配置摄像头数据源格式：分辨率、RGB/I420等*/
void CCaptureClass::ConfigCameraPin(HWND hwndParent)
{
    HRESULT hr;                              //返回值
    IAMStreamConfig *pSC;                    //流配置接口
    ISpecifyPropertyPages *pSpec;            //属性页接口
 //只有停止后，才能进行引脚属性的设置
 m_pMC->Stop();
//首先查询捕获CAPTURE、视频Video接口
 hr = m_pCapture->FindInterface(&PIN_CATEGORY_CAPTURE,
                          &MEDIATYPE_Video, m_pBF,
                          IID_IAMStreamConfig, (void **)&pSC);
CAUUID cauuid;                               //所有属性页结构体
 hr = pSC->QueryInterface(IID_ISpecifyPropertyPages, (void **)&pSpec);
 if(hr == S_OK)
 {
    hr = pSpec->GetPages(&cauuid);           //获取所有属性页
    //显示属性页
    hr = OleCreatePropertyFrame(hwndParent, 30, 30, NULL, 1,
                          (IUnknown **)&pSC, cauuid.cElems,
                          (GUID *)cauuid.pElems, 0, 0, NULL);
    //释放内存、资源
    CoTaskMemFree(cauuid.pElems);
    pSpec->Release();
    pSC->Release();
 }
 //回复运行
 m_pMC->Run();
}
```

上述代码首先以视频捕获方式获取流配置接口，在该接口下查询属性页接口，然后获取所有属性页，接着显示获取的属性页，最后释放资源。在显示了属性页后用户就可以对其属性页的对话框进行设置了，运行后的结果如图 9-1 所示。

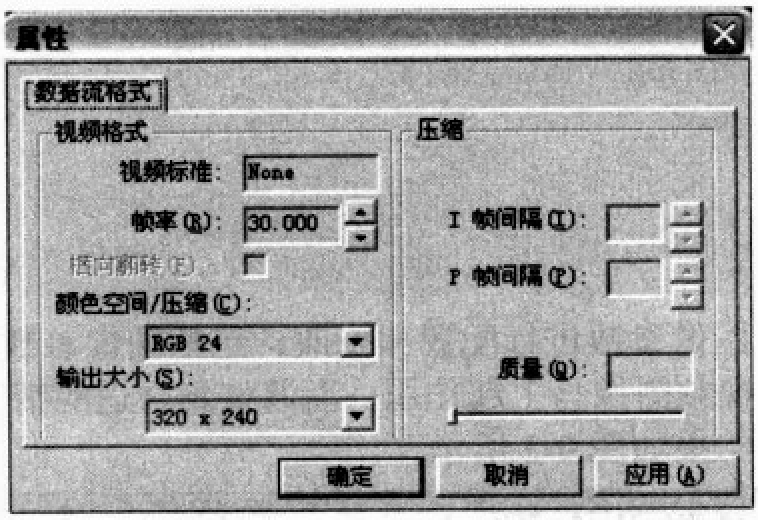


图 9-1 视频格式属性设置

2. 图像参数设置

图像参数设置类似于视频格式配置，根据设备的驱动类型 **VFW**、**WDM** 使用不同的技术配置设备的参数，这里我们同样假定采集设备的类型为 **WDM**，配置过程如下。

```
/*配置图像参数：亮度、色度、饱和度等*/
void CCaptureClass::ConfigCameraFilter(HWND hwndParent)
{
 HRESULT hr=0;
 ISpecifyPropertyPages *pProp;
 hr = m_pBF->QueryInterface(IID_ISpecifyPropertyPages, (void **)&pProp);
 if (SUCCEEDED(hr))
 {
        //获取滤波器名称和IUnknown接口指针
        FILTER_INFO FilterInfo;
        hr = m_pBF->QueryFilterInfo(&FilterInfo);
        IUnknown *pFilterUnk;
        m_pBF->QueryInterface(IID_IUnknown, (void **)&pFilterUnk);
        //显示该页
        CAUUID caGUID;
        pProp->GetPages(&caGUID);
             OleCreatePropertyFrame(
             hwndParent,              //父窗口
             0, 0,                    //Reserved
             FilterInfo.achName,      //对话框标题
             1,                       //该滤波器的目标数目
             &pFilterUnk,             //目标指针数组
             caGUID.cElems,           //属性页数目
             caGUID.pElems,           //属性页的CLSID数组
             0,                       //本地标识
             0, NULL                  //Reserved
        );
        //释放内存、资源
        CoTaskMemFree(caGUID.pElems);
```

```
        pFilterUnk->Release();
        FilterInfo.pGraph->Release();
        pProp->Release();
    }
    m_pMC->Run();
}
```

捕获滤波器属性是对图像参数进行配置和修改，包括图像参数、白平衡和模式控制。这些参数一旦更改，预览的视频图像马上起作用。当调整到合适的情况下，单击“确定”按钮，关闭该对话框，运行后的结果如图 9-2 所示。

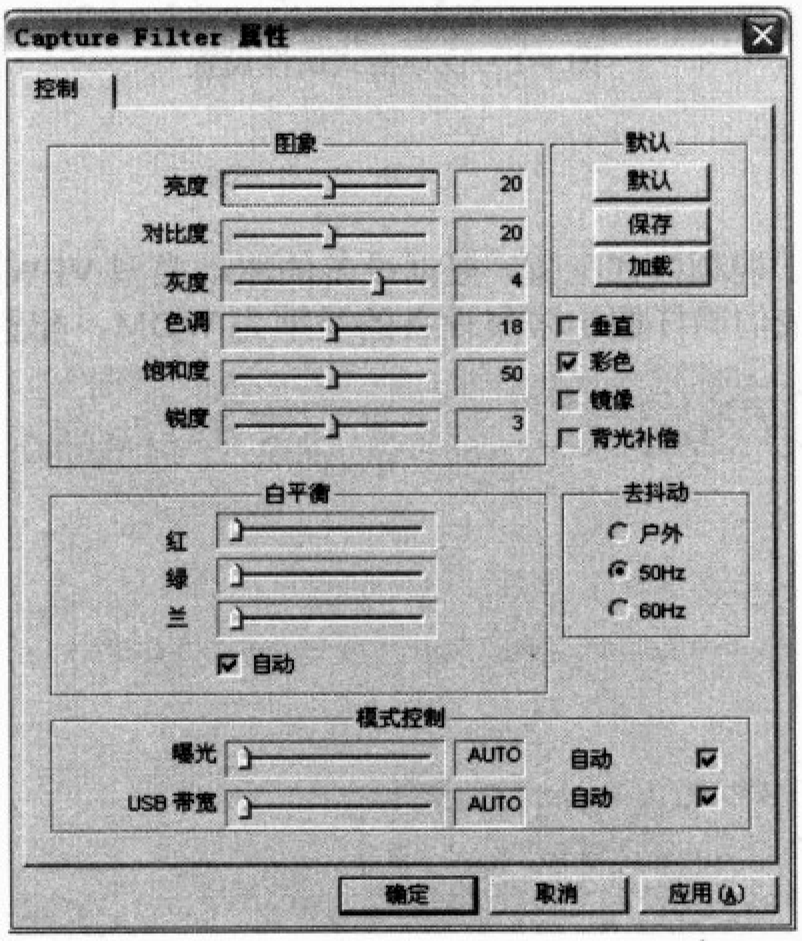


图 9-2　运行后的结果

9.2　使用经典采集技术实现视频捕获

DirectShow SDK 的视频采集经典技术是使用 ICaptureGraphBuilder2 标准接口，利用其方法 RenderStream 自动建立、连接滤波器链表。RenderStream 方法在预览、捕获视频时引脚的类型分为 PIN_CATEGORY_PREVIEW 和 PIN_CATEGORY_CAPTURE，媒体类型均为 MEDIATYPE_Video。此实例要完成的目的有两个：一是实时预览采集的视频数据；二是在预览图像的同时，实时地把捕获数据保存到文件中。首先我们使用 GraphEdit 模拟实现该过程。

9.2.1　GraphEdit模拟实现

STEP 01　添加“Video Capture Sources”视频捕获设备，如图 9-3 所示。

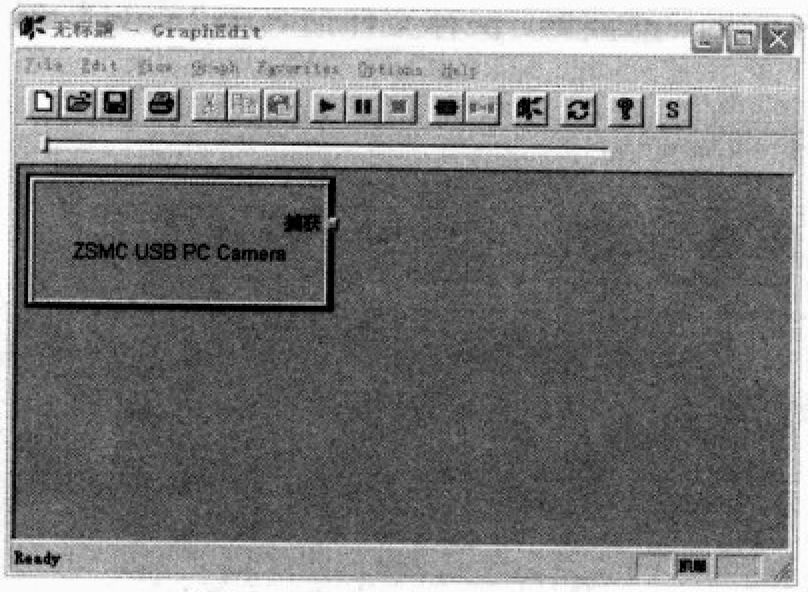


图 9-3　添加视频捕获设备

STEP 02 视频捕获滤波器只有一个 Pin，而我们要求在预览数据的同时还能够保存数据，即需要一个组件把捕获的流分成两个。DirectShow SDK 为此提供了 Smart Tee 滤波器，把捕捉的视频流分成两个流供使用。在 GraphEdit 中单击“DirectShow Filters”按钮，插入“Smart Tee”滤波器，如图 9-4 所示。

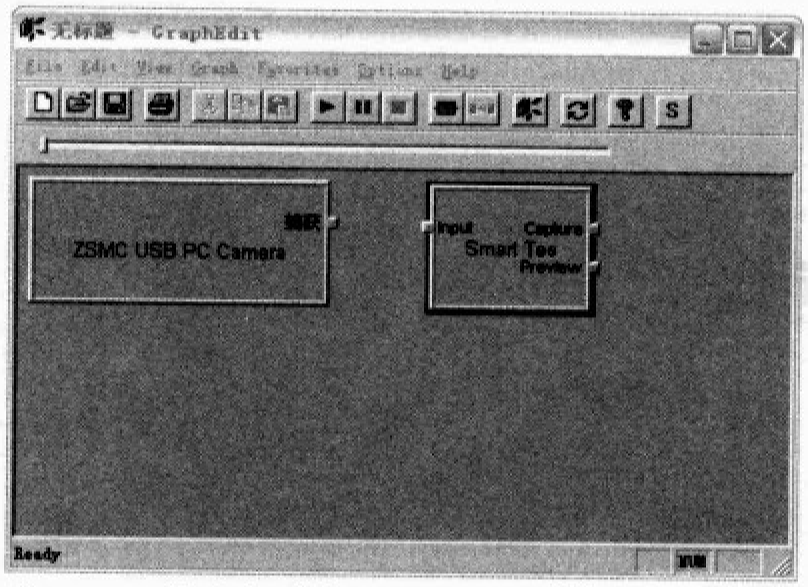


图 9-4　添加 Smart Tee 滤波器

STEP 03 采集捕捉的视频数据保存到文件，以 AVI 格式写文件。插入“AVI Mux”滤波器，如图 9-5 所示。

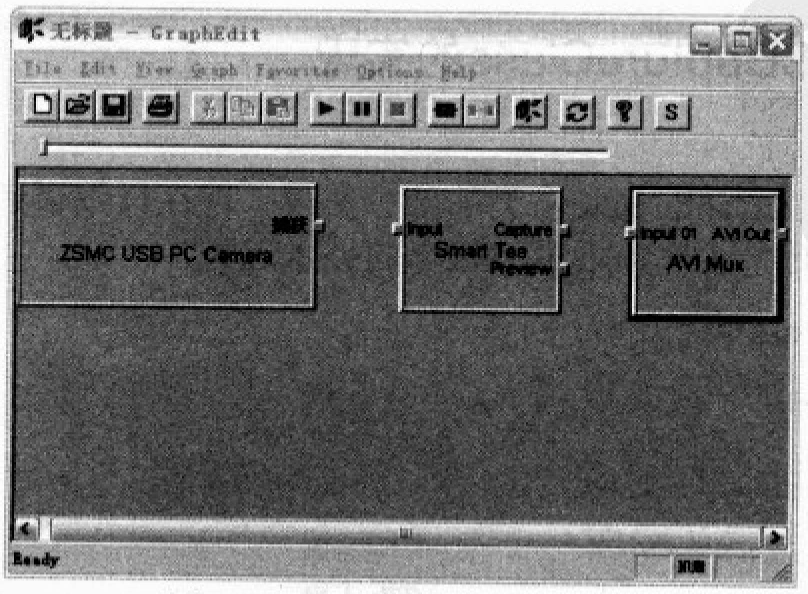


图 9-5　添加 AVI Mux 滤波器

STEP 04 插入“File writer”滤波器，保存文件命名为“a.avi”。然后插入“Video Renderer”滤波器，如图 9-6 所示。

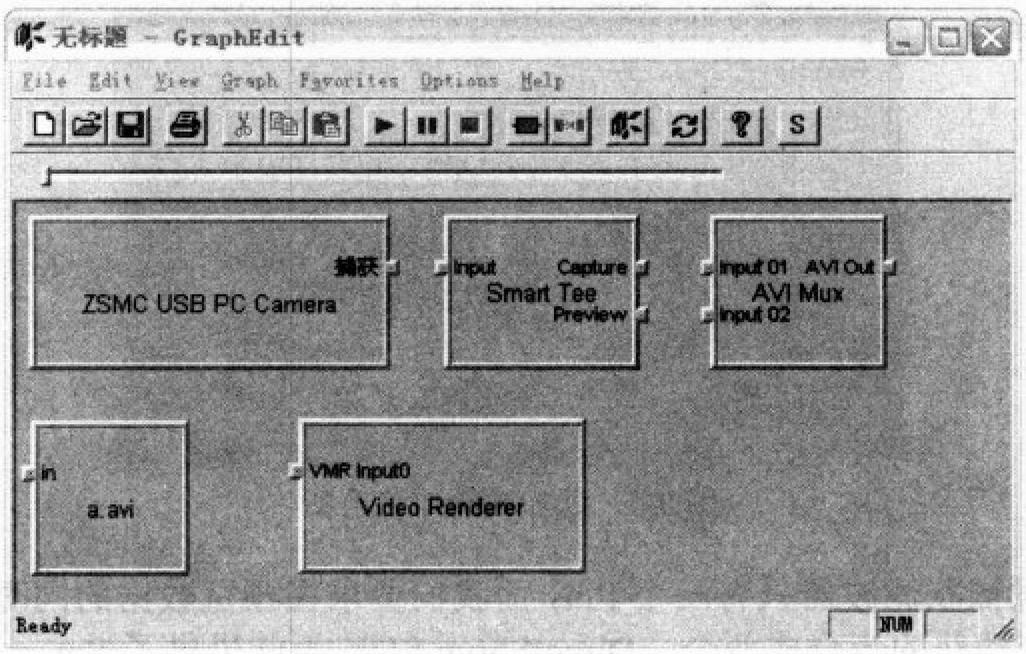


图 9-6 插入 File writer 和 Video Renderer 滤波器

STEP 05 最后把所有的滤波器用鼠标连接起来，完成构建滤波器链表，如图 9-7 所示。

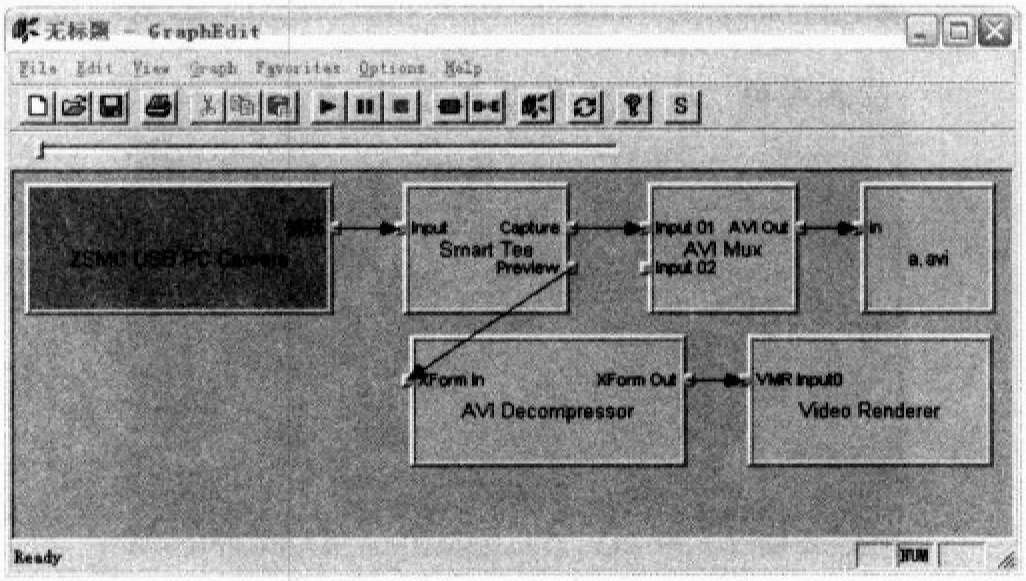


图 9-7 视频预览、保存滤波器链表

STEP 06 运行滤波器链表，单击“Graph”→“Play”按钮执行视频数据的预览、保存。

9.2.2 视频捕获类CCaptureClass的实现

在 9.1.2 节中讲述了 CCaptureClass 类的部分实现，现在详细讲述该类的成员变量和其他成员方法的实现，剖析其完成视频采集、保存的技术过程。

1）定义 CCaptureClass 类

```
class CCaptureClass
{
public:
    CCaptureClass();                //类构造器
    virtual ~CCaptureClass();       //类析构器
```

```
    int EnumDevices(HWND hList);
    void SaveGraph(TCHAR *wFileName);                    //保存滤波器链表
    void ConfigCameraPin(HWND hwndParent);               //配置摄像头的视频格式
    void ConfigCameraFilter(HWND hwndParent);            //配置摄像头的图像参数
    HRESULT CaptureImages(CString inFileName);           //捕获保存视频
    HRESULT PreviewImages(int iDeviceID, HWND hWnd);     //采集预览视频
private:
    HWND                      m_hWnd;          //视频显示窗口的句柄
    IGraphBuilder             *m_pGB;          //滤波器链表管理器
    ICaptureGraphBuilder2     *m_pCapture;     //增强型捕获滤波器链表管理器
    IBaseFilter               *m_pBF;          //捕获滤波器
    IMediaControl             *m_pMC;          //媒体控制接口
    IVideoWindow              *m_pVW;          //视频显示窗口接口
    IBaseFilter               *pMux;           //写文件滤波器
protected:
 bool BindFilter(int deviceId, IBaseFilter **pFilter);
                                      //把指定的设备滤波器捆绑到链表中
 void ResizeVideoWindow();            //更改视频显示窗口
 HRESULT SetupVideoWindow();          //设置视频显示窗口的特性
 HRESULT InitCaptureGraphBuilder();   //创建滤波器链表管理器，查询其各种控制接口
};
```

上述代码是类CCaptureClass的成员变量和成员函数，成员变量包括DirectShow开发流媒体播放应用程序需要的各种接口指针变量。成员函数实现创建滤波器链表管理器、配置视频采集格式、配置图像参数和保存滤波器链表等。

在类的构造器和析构器中完成对COM库的初始化和卸载，以及接口指针的初始化和资源释放。

```
/*定义的资源释放操作宏*/
#ifndef srelease
#define srelease(x)
if ( NULL != x )
{
  x->Release( );
  x = NULL;
}
#endif
/*类构造函数实现*/
CCaptureClass::CCaptureClass()
{
 CoInitialize(NULL);      //COM 库初始化
m_hWnd = NULL;            //视频显示窗口的句柄
m_pVW = NULL;             //视频窗口接口指针清空
m_pMC = NULL;             //媒体控制接口指针清空
m_pGB = NULL;             //滤波器链表管理器接口指针清空
m_pBF = NULL;             //捕获滤波器接口指针清空
m_pCapture = NULL;        //增强型捕获滤波器链表管理器接口指针清空
```

```
}
/*析构函数实现*/
CCaptureClass::~CCaptureClass()
{
 if (m_pMC)  m_pMC->Stop();                   //首先停止媒体
 if (m_pVW) {
       m_pVW->put_Visible(OAFALSE);           //视频窗口不可见
       m_pVW->put_Owner(NULL);                //视频窗口的父窗口清空
}
srelease(m_pCapture);                         //释放增强型捕获滤波器链表管理器接口
srelease(m_pMC);                              //释放媒体控制接口
srelease(m_pGB);                              //释放滤波器链表管理器接口
srelease(m_pBF);                              //释放捕获滤波器接口
CoUninitialize();                             //卸载COM库
}
```

在类构造函数中，清空所有指针以便于清楚其后续操作的状态。析构函数释放各种资源、指针并清空指针，最后卸载 COM 库。

2）根据指定的设备 ID，把基本滤波器与设备捆绑

首先枚举系统所有采集设备，直到列举的 ID 相同为止，最后 BindToObject 完成捆绑。

```
//把指定采集设备与滤波器捆绑
bool CCaptureClass::BindFilter(int deviceId, IBaseFilter **pFilter)
{
 if (deviceId < 0) return false;
 //枚举所有的视频捕获设备
 ICreateDevEnum *pCreateDevEnum;
//生成设备枚举器pCreateDevEnum
 HRESULT hr = CoCreateInstance(CLSID_SystemDeviceEnum, NULL,
                    CLSCTX_INPROC_SERVER,
                    IID_ICreateDevEnum,
                    (void**)&pCreateDevEnum);
 if (hr != NOERROR)  return false;
 IEnumMoniker *pEm;
//创建视频输入设备类枚举器
 hr = pCreateDevEnum->CreateClassEnumerator(CLSID_VideoInputDeviceCategory,
 &pEm, 0);
 if (hr != NOERROR) return false;
 pEm->Reset();          //复位该设备
 ULONG cFetched;
 IMoniker *pM;
 int index = 0;
//获取设备
 while(hr = pEm->Next(1, &pM, &cFetched), hr==S_OK, index <= deviceId)
 {
```

```
            IPropertyBag *pBag;
            //获取该设备的属性集
            hr = pM->BindToStorage(0, 0, IID_IPropertyBag, (void **)&pBag);
            if(SUCCEEDED(hr))
            {
                  VARIANT var;
                  var.vt = VT_BSTR;                         //保存的是二进制的数据
                  hr = pBag->Read(L"FriendlyName", &var, NULL);
                                                            //获取FriendlyName形式的信息
                  if (hr == NOERROR)
            {
                        //采集设备与捕获滤波器捆绑
                        if (index == deviceId) pM->BindToObject(0, 0, IID_IBaseFilter,
                        (void**)pFilter);
                        SysFreeString(var.bstrVal);     //释放二进制数据资源，必须释放
                  }
                  pBag->Release();
            }
            pM->Release();
            index++;
      }
      return true;
  }
```

该函数的传入参数是采集设备的索引号和捕获设备的滤波器。根据索引号查询系统中的视频捕获设备。以友好名字（FriendlyName）的方式获取所选设备的信息，然后把查询成功的设备与传入的滤波器捆绑，返回捕获设备的滤波器。

3）设置视频显示窗口

DirectShow 的显示窗口与 IVideoWindow 接口的设置基本相同，把传入的显示窗口的句柄捆绑到 IvideoWindow 接口上。

```
/*设置视频显示窗口的特性*/
HRESULT CCaptureClass::SetupVideoWindow()
{
 HRESULT hr;
 //m_hWnd为类CCaptureClass的成员变量，在使用该函数前须初始化
 hr = m_pVW->put_Visible(OAFALSE);                  //视频窗口不可见
 hr = m_pVW->put_Owner((OAHWND)m_hWnd);             //窗口所有者：传入的窗口句柄
 if (FAILED(hr)) return hr;
 hr = m_pVW->put_WindowStyle(WS_CHILD | WS_CLIPCHILDREN);//设置窗口类型
 if (FAILED(hr)) return hr;
 ResizeVideoWindow();                                       //更改窗口大小
 hr = m_pVW->put_Visible(OATRUE);                           //视频窗口可见
 return hr;
}
```

```
/*更改视频窗口大小*/
void CCaptureClass::ResizeVideoWindow()
{
 if (m_pVW) {
        //让图像充满整个指定窗口
        CRect rc;
        ::GetClientRect(m_hWnd,&rc);                    //获取显示窗口的客户区
        m_pVW->SetWindowPosition(0, 0, rc.right, rc.bottom);
        //设置视频显示窗口的位置
 }
}
```

在调用该函数前，需要把应用程序的显示窗口句柄传入以初始化 m_hWnd。首先视频窗口不可见，捆绑传入的窗口句柄为视频窗口，设置窗口类型。ResizeVideoWindow 函数获取显示窗口的客户区域，利用视频窗口接口的方法 SetWindowPosition 设置视频显示窗口的位置。

4）预览采集到的视频数据

使用上述有关的类成员函数初始化滤波器链表管理器，把指定采集设备的滤波器添加到链表中，然后渲染 RenderStream 方法把所有的滤波器链接起来，最后根据设定的显示窗口预览采集到的视频数据，具体实现过程如下。

```
/*开始预览视频数据*/
HRESULT CCaptureClass::PreviewImages(int iDeviceID, HWND hWnd)
{
 HRESULT hr;
 //初始化视频捕获滤波器链表管理器
 hr = InitCaptureGraphBuilder();
 if (FAILED(hr)) {
        AfxMessageBox(_T("Failed to get video interfaces!"));
        return hr;
 }
 //把指定采集设备与滤波器捆绑
 if (!BindFilter(iDeviceID, &m_pBF))   return S_FALSE;
 //把滤波器添加到滤波器链表中
 hr = m_pGB->AddFilter(m_pBF, L"Capture Filter");
//渲染媒体，把链表中滤波器链接起来
hr = m_pCapture->RenderStream( &PIN_CATEGORY_PREVIEW,
                               &MEDIATYPE_Video, m_pBF, NULL, NULL );
 if( FAILED( hr ) ) {
        AfxMessageBox(_T("Can't build the graph"));
        return hr;
 }
 //设置视频显示窗口
 m_hWnd = hWnd;             //初始化窗口句柄
 SetupVideoWindow();        //设置显示窗口
```

```
    hr = m_pMC->Run();          //开始采集、预览视频，在指定窗口显示视频
    if(FAILED(hr)) {
          AfxMessageBox(_T("Couldn't run the graph!"));
          return hr;
    }
    return S_OK;
}
```

上述程序从最初的创建滤波器链表管理器、枚举系统视频采集设备、把采集设备与捕获滤波器捆绑，到添加滤波器到滤波器链表、设置视频显示窗口，最后开始运行媒体：采集、预览视频，包含了使用 DirectShow SDK 开发视频采集、预览的整个技术过程。函数功能独立而又完整。

5）保存采集到的数据

把捕获的视频以 AVI 格式写文件。注意设置前停止调用滤波器链表，设置完成后再运行链表。

```
    /*设置捕获视频的文件，开始捕捉视频数据写文件*/
HRESULT CCaptureClass::CaptureImages(CString inFileName)
{
 HRESULT hr=0;
m_pMC->Stop();                              //先停止视频
//设置文件名，注意第二个参数的类型
 hr = m_pCapture->SetOutputFileName( &MEDIASUBTYPE_Avi,
                                      inFileName.AllocSysString(), &pMux, NULL );
  //渲染媒体，链接所有滤波器
 hr = m_pCapture->RenderStream( &PIN_CATEGORY_CAPTURE,
                                       &MEDIATYPE_Video, m_pBF, NULL, pMux );
 pMux->Release();
 m_pMC->Run();                              //回复视频
 return hr;
}
```

预览视频后，用户可以使用该函数存储捕获的视频数据。首先停止视频媒体，利用 ICaptureGraphBuilder2 的方法 SetOutputFileName 设置存储捕获数据的文件名，然后渲染视频媒体，RenderStream 方法自动链接图表中的滤波器，最后开始运行视频媒体。

9.2.3 界面设计

本案例使用 VC++ 2005 的对话框应用程序框架设计视频捕获应用程序。界面设计步骤与前面第 8 章的界面设计基本相同，这里不再列出 VC++ 2005 应用程序向导建立对话框程序框架的过程。下面详细列出该案例设计的具体步骤。

STEP 01 应用 VC++ 2005 应用程序向导建立对话框程序框架，项目名称为 CaptureVideo。

STEP 02 在项目 CaptureVideo 的主界面中添加控件：6 个 Button、1 个 Combo Box、1

个 Picture Control，如表 9-1 所示。根据其功能修改所有控件的 ID。

表 9-1　控件功能列表

名称	说明
ID_PREVIEW	视频预览
ID_CAPTURE	视频捕获
ID_VIDEO_FORMAT	视频格式
ID_IMAGE_PARAMETER	图像参数
ID_SAVEGRAPH	保存图表
ID_EXIT	退出程序（终止预览捕获）
IDC_DEVICE_LISTER	设备列表组合框
IDC_VIDEO_WINDOW	显示捕获的视频图像

STEP 03 在添加所有的控件后，修改控件大小，重排其位置。添加控件后的主界面设计如图 9-8 所示。

STEP 04 控件与变量捆绑。为便于控制，把 IDC_VIDEO_WINDOW 控件、IDC_DEVICE_LISTER 控件分别与变量捆绑。途径是选择控件的右键菜单中的“添加变量”命令，如图 9-9 所示，激活“欢迎使用添加成员变量向导”。

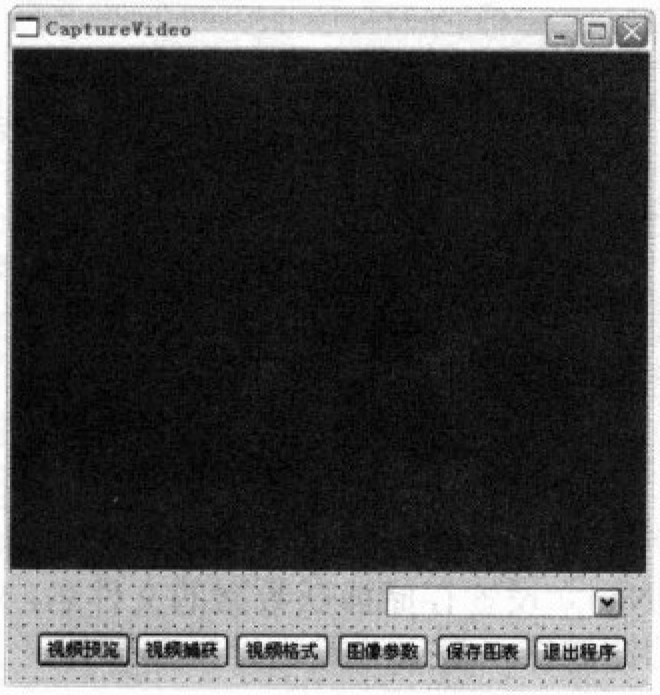


图 9-8　采集视频界面设计

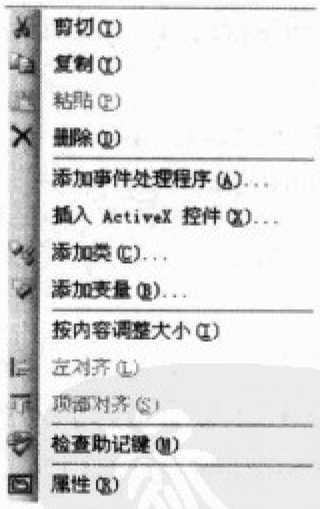


图 9-9　控件右键菜单内容

添加控件变量后的 CCaptureVideoDlg 类的代码如下。

```
//显示捕获的图像
CStatic m_videoWindow;
//组合框列表，显示设备名称
CComboBox m_listCtrl;
```

STEP 05 添加按钮在线提示 ToolTip。

首先在类 CCaptureVideoDlg 定义中声明 tooltip 控件。

```
CToolTipCtrl m_tooltip;
```

接着在类 CCaptureVideoDlg 实现文件的对话框初始化函数 OnInitDialog 中添加：

```
m_tooltip.Create(this);
m_tooltip.Activate(TRUE);
m_tooltip.AddTool(GetDlgItem(ID_PREVIEW), _T("开始预览视频"));
    //添加其他按钮的tooltip
```

在 PreTranslateMessage 消息处理函数中添加如下代码，如果程序中没有该消息处理函数，则需要用户自己添加。添加过程同 8.4.1 小节。

```
m_tooltip.RelayEvent(pMsg);
```

STEP 06 功能按钮的消息响应，即单击按钮的事件处理。

双击某按钮，实现单击按钮事件处理函数的添加。为了使用类 CCaptureClass 的变量和函数，在 CCaptureVideoDlg 类中引入头文件并定义视频捕获类的对象。

首先引入头文件：

```
#include "CaptureClass.h"
```

接着在 CCaptureVideoDlg 类中定义视频捕获类的对象：

```
CCaptureClass m_cap;
```

双击“视频预览”按钮，添加事件处理代码：

```
void CCaptureVideoDlg::OnBnClickedPreview()
{
 //TODO: 在此添加控件通知处理程序代码
 HWND hVWindow = m_videoWindow.GetSafeHwnd();      //获取视频显示窗口的句柄
 int id = m_listCtrl.GetCurSel();                  //获取当前选中的视频设备
 m_cap.PreviewImages(id , hVWindow);               //开始预览视频
}
```

获取视频显示窗口的句柄，根据程序最初枚举到的所有采集设备，选择用户选中的采集设备。调用视频捕获类的成员函数 PreviewImages 完成整个视频的采集、显示任务。图 9-10 所示为“视频预览”按钮实现的效果图。

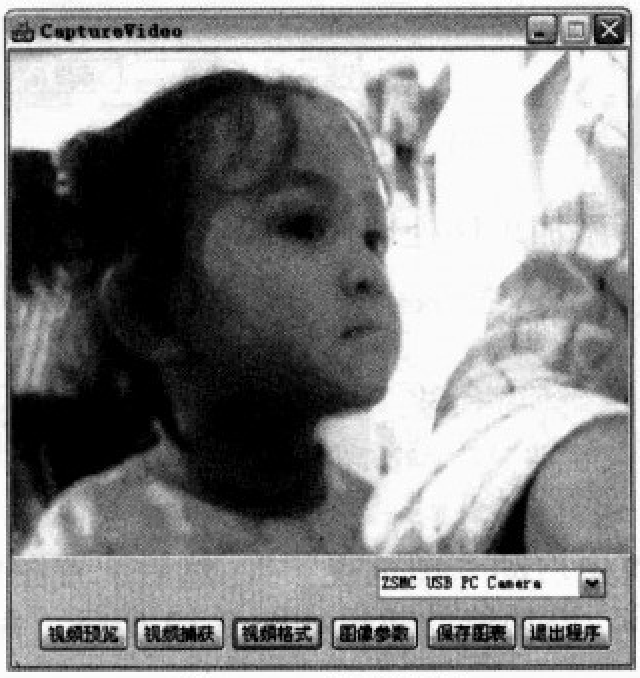


图 9-10　视频预览

双击“视频捕获”按钮，添加事件处理代码。

```
void CCaptureVideoDlg::OnBnClickedCapture()
{
 //TODO: 在此添加控件通知处理程序代码
 CString strFilter = _T("AVI File (*.avi) | *.avi|"); //文件类型过滤器
 strFilter += "All File (*.*) | *.*|";
 CFileDialog dlg(TRUE, NULL, NULL,                        //打开另存为文件对话框
                   OFN_PATHMUSTEXIST|OFN_HIDEREADONLY, strFilter, this);
 if (dlg.DoModal() == IDOK) {
         CString m_sourceFile = dlg.GetPathName();      //获取用户输入的文件路径名
         m_cap.CaptureImages(m_sourceFile);             //开始捕获、存储视频
 }
}
```

存储捕获的视频，这里采用 AVI 格式，视频数据未经压缩。调用视频捕获类 CCaptureClass 的成员函数 CaptureImages 完成视频的捕获、存储任务。本案例需要先预览视频，然后再开始捕获、存储视频，不能直接捕获、存储视频。双击“视频格式”按钮，添加事件处理代码。

```
void CCaptureVideoDlg::OnBnClickedVideoFormat()
{
 //TODO: 在此添加控件通知处理程序代码
 m_cap.ConfigCameraPin(this->m_hWnd);
}
```

设置视频格式前，首先启动“预览视频”，然后再配置视频的格式。视频格式包括帧率、颜色空间、视频分辨率等，该功能的效果如图 9-11 所示。

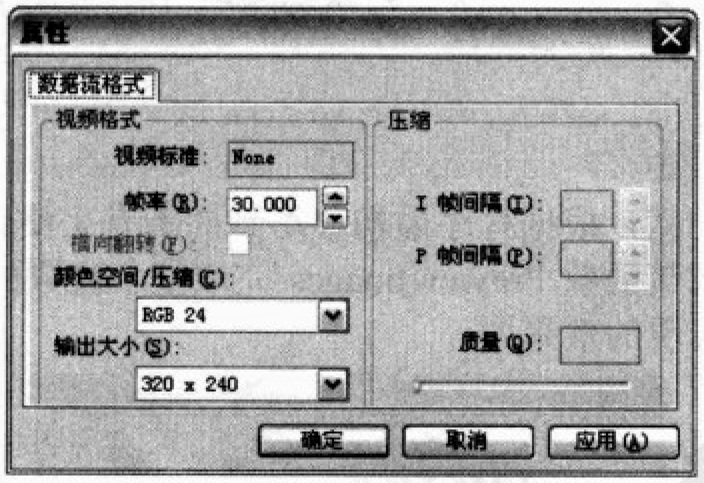


图 9-11　视频格式配置属性页

双击“视频参数”按钮，添加事件处理代码。

```
void CCaptureVideoDlg::OnBnClickedImageParameter()
{
 //TODO: 在此添加控件通知处理程序代码
 m_cap.ConfigCameraFilter(this->m_hWnd);
}
```

设置视频图像参数前，首先启动“预览视频”，然后配置图像参数。图像参数包括图像、白平衡、模式控制和去抖动等，该功能的效果如图 9-12 所示。并且该功能调整的参数马上起作用。

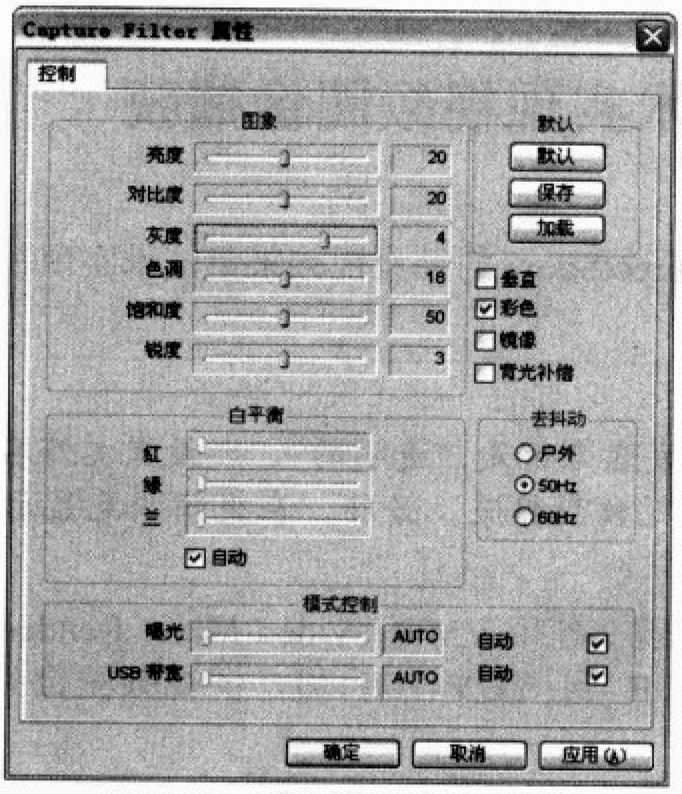


图 9-12　图像参数配置属性页

双击“保存图表”按钮，添加事件处理代码。

```
void CCaptureVideoDlg::OnBnClickedSavegraph()
{
 //TODO: 在此添加控件通知处理程序代码
 CFileDialog dlg(TRUE);
 if (dlg.DoModal()==IDOK) {
        CString str=dlg.GetPathName();              //要保存的Graph文件名
        TCHAR *inFileName = str.GetBuffer(str.GetLength());//获取字符串指针
        str.ReleaseBuffer();                        //切记要释放Buffer
        m_cap.SaveGraph(inFileName);                //保存Graph
 }
}
```

本功能存储的 Graph 文件可以使用程序 GraphEdit 播放。双击“退出程序”按钮，添加事件处理代码。程序隐含调用了类 CCaptureClass 的析构函数，释放资源和 COM 库。

```
void CCaptureVideoDlg::OnBnClickedExit()
{
 //TODO: 在此添加控件通知处理程序代码
 CDialog::OnOK();
}
```

退出本程序时，由于视频捕获类 CCaptureClass 的析构函数已经包含了释放资源、指针的工作，所以退出应用程序时不用释放任何资源，只是关闭应用程序。

STEP 07 其他工作。

在对话框的初始化 OnInitDialog 中枚举本系统的视频采集设备，添加到列表框并默认显示第一个设备。

```
m_cap.EnumDevices(m_listCtrl.GetSafeHwnd());
m_listCtrl.SetCurSel (0);
```

9.3 使用VMR技术实现视频图像捕获

以上实例展示了使用 DirectShow 的经典视频采集技术实现视频的预览、捕获的过程。

注意：

文件的保存对我们开发者来说是“透明的”，即设置文件名后启动开始预览保存。开发者不直接访问采集到的视频数据流。另外，在视频预览显示时，图像有抖动、不流畅的现象。

在下面的实例中我们实现在使用 VMR（Video Mixer Render）流畅预览视频的同时捕捉图像帧，转换颜色空间 RGB 到 YUV420/I420，供视频图像处理算法处理，或直接保存 RGB/YUV 格式文件。

9.3.1 VMR技术基础

前面显示图像时使用的是 Video Renderer 视频渲染技术，可应用于所有支持 DirectX 的平台，是视频显示较早的技术应用，对平台没有特殊要求；在 Windows XP 操作系统之前是默认的显示技术。

视频混合渲染 Video Mixing Renderer 7（VMR-7）可用于 WindowsXP 操作系统，且是该系统的默认选择，与旧的显示滤波器相比功能有大的提升。VMR-9 是 DirectShow SDK 9.0 中的视频渲染技术，采用 Direct3D 技术。VMR-9 不是默认选择的滤波器，对系统要求更高，但功能更强、效果更好，它使用了最新的图像 API 函数提供了最好的显示性能。

视频显示根据有无窗口可分为有窗口模式和无窗口模式。Video Renderer 只支持窗口模式，VMR 支持有窗口（Window）和无窗口（Windowless）两种模式，默认支持有窗口模式。在无窗口模式中，可以把视频和应用程序主界面的窗口捆绑。

VMR 在清晰、流畅显示视频的同时，可以灵活地捕获图像帧，这也是我们选择 VMR 技术的一个重要原因，因为我们想直接对图像数据进行处理。

9.3.2 GraphEdit模拟实现视频捕获预览

同样，在编程使用 VMR 技术实现视频预览、捕获前，利用微软的 GraphEdit 程序快速构建 VMR 的滤波器链表，感受 VMR 的处理效果。

GraphEdit 模拟实现视频捕获预览的具体过程如下。

STEP 01 启动 GraphEdit 应用程序，选择插入“Video Capture Sources”滤波器，如图 9-13 所示。

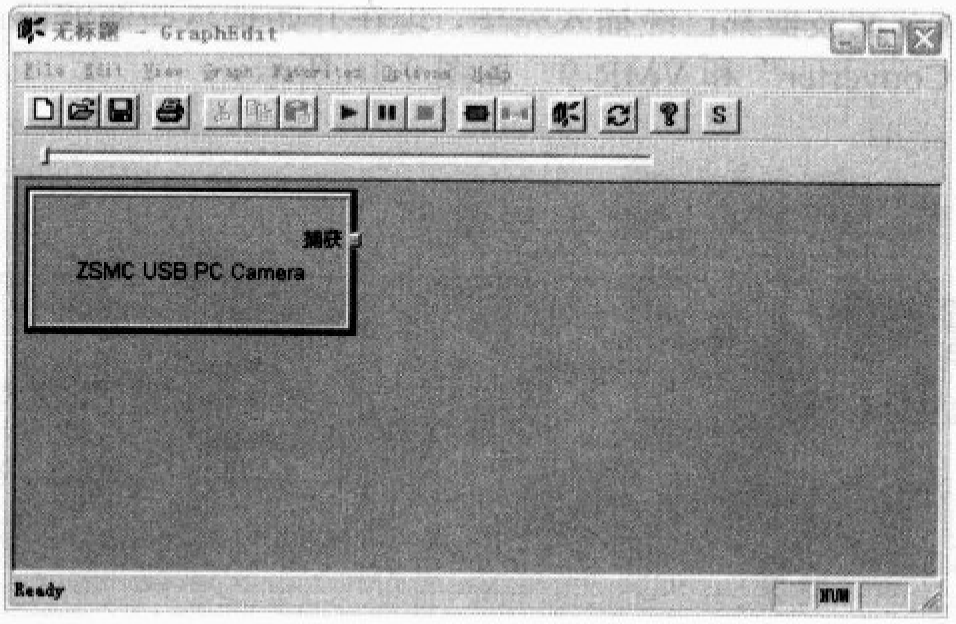


图 9-13 视频源滤波器

STEP 02 由于 VMR-9 显示的数据格式是根据系统显卡的支持格式确定的，如ARGB32/YUV2 等，所以如果采集设备不支持这些类型就需要在采集设备和 VMR 之间加一个转换滤波器。根据作者测试，有两种转换滤波器可担当媒体类型的匹配转换任务：AVI Decompressor 和 Color Space Converter。选择插入“AVI Decompressor”和“Color Space Converter”滤波器，如图 9-14 所示。

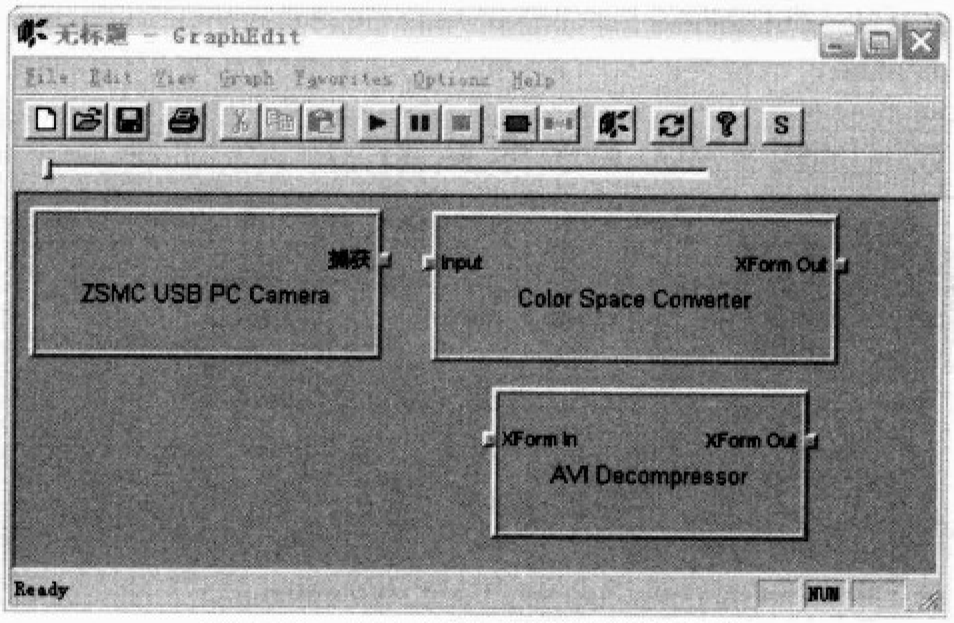


图 9-14 插入媒体类型匹配滤波器

STEP 03 选择插入“Video Mixing Render 9”滤波器，如图 9-15 所示。

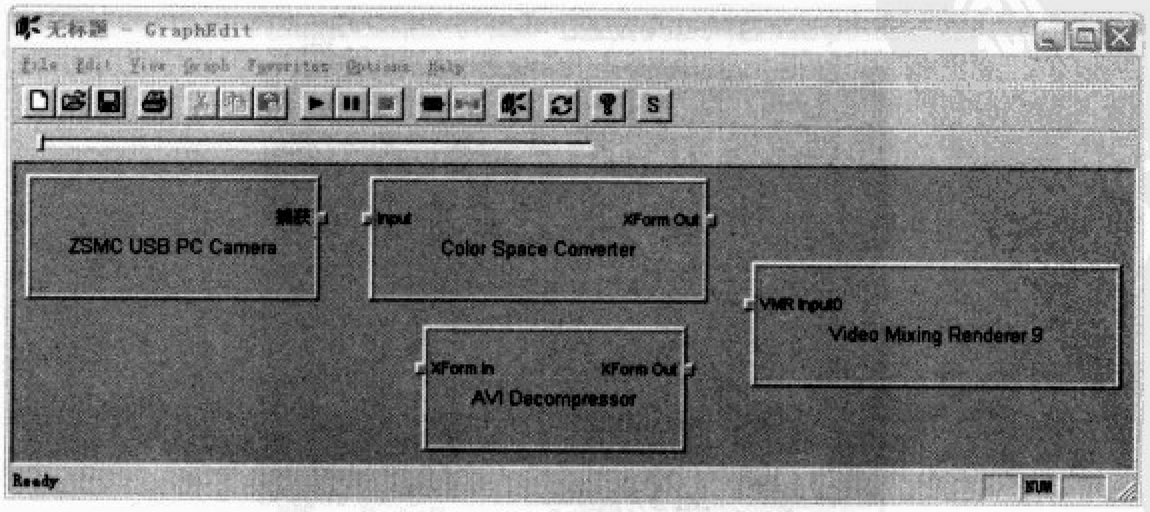


图 9-15 插入 VMR-9 滤波器

STEP 04 至此所有滤波器都已经插入完毕，现在开始把这些滤波器链接起来。链接源滤波器、“Color Space Converter”和 VMR-9，如图 9-16 所示。

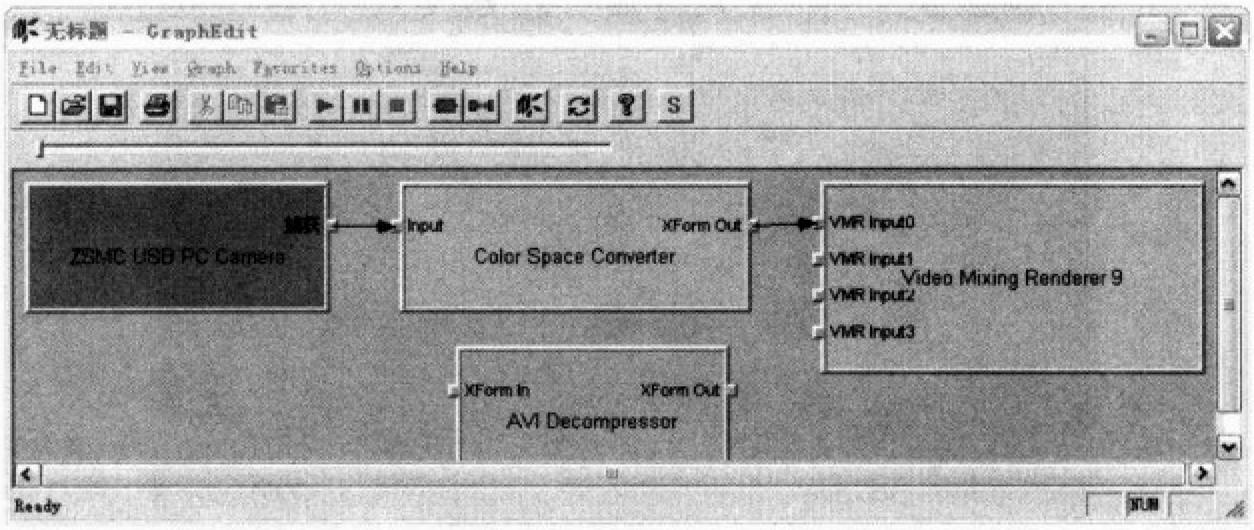


图 9-16 GraphEdit 工具

STEP 05 单击工具栏上的 ▶ 按钮，开始运行该滤波器链表，预览视频数据。如果没有出现预览视频窗口，则请首先确定视频设备是否安装或正常工作，然后再运行。

STEP 06 单击 按钮，断开链接。链接源滤波器、“AVI Decompressor”和 VMR-9，如图 9-17 所示。注意链接的顺序是从左至右，即“Source Filter”→“AVI Decompressor”→“VMR-9”，否则 GraphEdit 会报错。

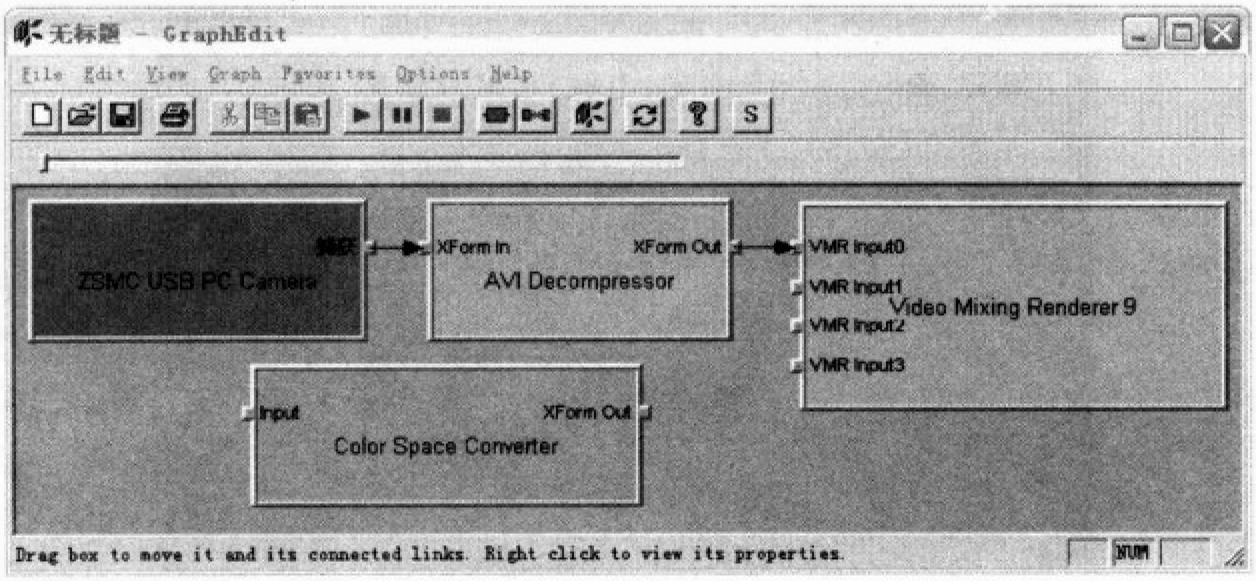


图 9-17 链接源滤波器、AVI Decompressor 和 VMR-9

STEP 07 单击工具栏上的 ▶ 按钮，运行链表，开始预览视频，如图 9-18 所示。

图 9-18 使用 VMR 技术预览视频数据

9.3.3 视频图像捕获类CVMR_Capture的实现

为完成采集视频、预览视频和捕获图像帧的任务，我们封装了 CVMR_Capture 类，使用 VMR 技术流畅显示和捕捉图像帧。根据 GraphEdit 的滤波器内容和链表的操作，编程实现使用 VMR 技术显示、捕捉图像。

STEP 01 类 CVMR_Capture 定义在头文件 VMR_Capture.h 中。

```
enum PLAYER_STATE {INIT,RUNNING,PAUSED,STOPPED};
class CVMR_Capture
{
public:
 CVMR_Capture();                                        //类构造器
 virtual ~CVMR_Capture();                               //类析构器
 int EnumDevices(HWND hList);                           //枚举设备
 HRESULT Init(int iDeviceID,HWND hWnd,int iWidth,int iHeight);
 //根据设备索引号初始化
 DWORD GetFrame(BYTE ** pFrame);                        //获取捕获的图像帧
 BOOL Pause();                                          //暂停预览、捕获
 DWORD GrabFrame();                                     //截获图像
 void CloseInterfaces(void);                            //关闭接口，释放资源
 void SaveGraph(CString wFileName);                     //保存滤波器链表
protected:
 IGraphBuilder            *m_pGB;                       //滤波器链表管理器
 IMediaControl            *m_pMC;                       //媒体控制接口
 IMediaEventEx            *m_pME;                       //媒体事件接口
 IVMRWindowlessControl9   *m_pWC;                       //VMR-9接口
 IPin                     *m_pCamOutPin;                //视频采集滤波器引脚
 IBaseFilter              *m_pDF;                       //视频采集滤波器
 PLAYER_STATE             m_psCurrent;
 int      m_nWidth;                                     //图像帧宽度
 int      m_nHeight;                                    //图像帧高度
 BYTE     *m_pFrame;                                    //捕获的图像帧数据指针
 long     m_nFramelen;                                  //捕获的图像帧数据大小
 bool BindFilter(int deviceId, IBaseFilter **pFilter);        //设备与滤波器捆绑
 HRESULT InitializeWindowlessVMR(HWND hWnd); //初始化无窗口模式的VMR
 HRESULT InitVideoWindow(HWND hWnd,int width, int height);    //初始化视频窗口
 void StopCapture();
 void DeleteMediaType(AM_MEDIA_TYPE *pmt);                    //删除媒体类型
 bool Convert24Image(BYTE *p32Img,BYTE *p24Img,DWORD dwSize32); //颜色空间转换
private:
};
```

CVMR_Capture 类封装了使用 VMR 技术的成员函数和成员变量，包括类的构造器、析构器，滤波器链表管理器接口、媒体控制接口、媒体事件接口，图像帧的宽度、高度及操作

VMR 的保护成员函数等。

STEP 02 类 CVMR_Capture 有关初始化、析构等，类实现 VMR_Capture.cpp 文件中。

```
//释放资源的宏
#define RELEASE_POINTER (x) { if (x) x->Release(); x = NULL; }
/*类构造器*/
CVMR_Capture::CVMR_Capture()
{
 CoInitialize(NULL);                        //初始化COM库
//接口指针清空
 m_pGB = NULL;        m_pMC = NULL;
m_pME = NULL;         m_pWC = NULL;
 m_pDF =NULL;         m_pCamOutPin =NULL;
m_pFrame=NULL;        m_nFramelen=0;
 m_psCurrent=STOPPED;
}
/*类析构器*/
CVMR_Capture::~CVMR_Capture()
{
 CloseInterfaces();                         //清空指针、释放资源
 CoUninitialize( );                         //卸载COM库
}
/*释放所有资源，断开链接*/
void CVMR_Capture::CloseInterfaces(void)
{
      HRESULT hr;
       //停止媒体回放
      if(m_pMC) hr = m_pMC->Stop();
      m_psCurrent = STOPPED;
       //释放并清零接口指针
 if(m_pCamOutPin)
       m_pCamOutPin->Disconnect();           //断开引脚的链接
RELEASE_POINTER(m_pCamOutPin);              //视频采集滤波器引脚清空
RELEASE_POINTER(m_pMC);                     //媒体控制接口清空
RELEASE_POINTER(m_pGB);                     //滤波器链表管理器接口清空
RELEASE_POINTER(m_pWC);                     //VMR-9接口清空
RELEASE_POINTER(m_pDF);                     //视频采集滤波器接口清空
 //释放分配的内存
 if(m_pFrame!=NULL)
        delete []m_pFrame;                  //释放图像空间缓存
 }
```

上述程序完成资源释放、指针清空等工作，在退出应用程序时调用此函数。

STEP 03 枚举本地系统的采集设备。

```
/* 枚举本地系统的采集设备*/
int  CVMR_Capture::EnumDevices(HWND hList)
```

```
{
  if (!hList) return -1;
  int id = 0;
        //枚举所有视频采集设备
      ICreateDevEnum *pCreateDevEnum;
        //生成设备枚举器
  HRESULT hr = CoCreateInstance(CLSID_SystemDeviceEnum, NULL,
                         CLSCTX_INPROC_SERVER,
                         IID_ICreateDevEnum, (void**)&pCreateDevEnum);
   if (hr != NOERROR) return -1;
      IEnumMoniker *pEm;
      //创建视频类枚举器
   hr = pCreateDevEnum->CreateClassEnumerator(CLSID_VideoInputDeviceCategory,
   &pEm, 0);
   if (hr != NOERROR) return -1;
   pEm->Reset();                                          //复位设备
   ULONG cFetched;
   IMoniker *pM;
   while(hr = pEm->Next(1, &pM, &cFetched), hr==S_OK)     //根据索引获取设备
   {
     IPropertyBag *pBag;
     //获取属性集
     hr = pM->BindToStorage(0, 0, IID_IPropertyBag, (void **)&pBag);
     if(SUCCEEDED(hr))
     {
         VARIANT var;
         var.vt = VT_BSTR;                                //保存二进制数据
         hr = pBag->Read(L"FriendlyName", &var, NULL);
         //以FriendlyName获取设备信息
         if (hr == NOERROR)
         {
              id++;
              //把该设备的信息添加到组合框
              (long)SendMessage(hList, CB_ADDSTRING, 0,(LPARAM)var.
              bstrVal);
              SysFreeString(var.bstrVal);                 //释放资源
          }
          pBag->Release();                                //释放资源
     }
     pM->Release();                                       //释放资源
   }
 return id;
}
```

上述程序实现枚举系统所有的视频采集设备，然后把 FriendlyName 信息添加到组合框中，这里的设备枚举与“使用经典采集技术实现视频捕获”实例的设备枚举相同。

STEP 04 构建滤波器链表，添加各个滤波器、链接并运行链表。应用程序调用该函数实现视频采集、图像预览。

```
HRESULT CVMR_Capture::Init(int iDeviceID,HWND hWnd, int iWidth, int iHeight)
{
    HRESULT hr;
    //再次调用函数，释放已经建立的链表
    CloseInterfaces();
    //创建IGraphBuilder
   hr = CoCreateInstance(CLSID_FilterGraph, NULL,    CLSCTX_INPROC_SERVER,
                   IID_IGraphBuilder, (void **)&m_pGB);
   if (SUCCEEDED(hr))
   {
     //创建VMR并添加到Graph中
       InitializeWindowlessVMR(hWnd);
     //把指定的设备捆绑到一个滤波器
     if(!BindFilter(iDeviceID, &m_pDF))
           return S_FALSE;
     //添加采集设备滤波器到Graph中
     hr = m_pGB->AddFilter(m_pDF, L"Video Capture");
     if (FAILED(hr)) return hr;
     //获取捕获滤波器的引脚
     IEnumPins  *pEnum;
     m_pDF->EnumPins(&pEnum);
     hr |= pEnum->Reset();
     hr |= pEnum->Next(1, &m_pCamOutPin, NULL);
     //获取媒体控制和事件接口
        hr |= m_pGB->QueryInterface(IID_IMediaControl, (void **)&m_pMC);
        hr |= m_pGB->QueryInterface(IID_IMediaEventEx, (void **)&m_pME);
     //设置窗口通知消息处理
        //hr = pME->SetNotifyWindow((OAHWND)hWnd, WM_GRAPHNOTIFY, 0);
     //匹配视频分辨率，对视频显示窗口设置
     hr |= InitVideoWindow(hWnd,iWidth, iHeight);
     //为捕获图像帧申请内存
     m_nFramelen=iWidth*iHeight*3;
     m_pFrame=(BYTE*) new BYTE[m_nFramelen];
     //运行Graph,捕获视频
     m_psCurrent = STOPPED;
     hr |= m_pGB->Render(m_pCamOutPin);
     hr |= m_pMC->Run();
     if (FAILED(hr)) return hr;
     m_psCurrent = RUNNING;
  }
  return hr;
}
```

上述是一个独立而又完整的使用 **VMR-9** 技术实现视频的采集、存储任务的程序。首先

如果再次调用该函数，则关闭所有接口、释放有关资源；接着创建 IGraphBuilder 作为滤波器链表管理器；然后添加 VMR 滤波器到链表中，把指定的采集设备索引与捕获滤波器捆绑，并把该滤波器添加到链表中；接着获取捕获滤波器的引脚、获取媒体控制接口和事件接口；设置窗口通知消息处理，根据输入的视频分辨率匹配采集设备的分辨率；最后使用自动渲染功能 Render 方法把滤波器链表链接起来，使用媒体控制接口的方法 Run 开始运行媒体。以下分别是该函数的子功能实现。

STEP 05 创建 VMR 滤波器，并添加到 Graph 链表中。

```
/*创建VMR，添加、设置VMR*/
HRESULT CVMR_Capture::InitializeWindowlessVMR(HWND hWnd)
{
    IBaseFilter* pVmr = NULL;
    //创建VMR
    HRESULT hr = CoCreateInstance(CLSID_VideoMixingRenderer, NULL,
                        CLSCTX_INPROC, IID_IBaseFilter, (void**)&pVmr);
    if (SUCCEEDED(hr))
    {
     //添加VMR到滤波器链表中
        hr = m_pGB->AddFilter(pVmr, L"Video Mixing Renderer");
        if (SUCCEEDED(hr))
        {
            //设置无窗口渲染模式
            IVMRFilterConfig* pConfig;
            hr = pVmr->QueryInterface(IID_IVMRFilterConfig, (void**)&pConfig);
            if( SUCCEEDED(hr))
            {
                pConfig->SetRenderingMode(VMRMode_Windowless);
                pConfig->Release();
            }
            //设置传入的窗口为显示窗口
            hr = pVmr->QueryInterface(IID_IVMRWindowlessControl, (void**)&m_pWC);
            if( SUCCEEDED(hr))
            {
                m_pWC->SetVideoClippingWindow(hWnd);      //设置视频剪辑窗口
            }
        }
        pVmr->Release();
    }
    return hr;
}
```

首先使用 CoCreateInstance 创建 VMR 的接口 pVmr，然后把 VMR 滤波器添加到滤波器链表中。设置视频显示为无窗口模式，首先在 pVmr 接口下查询 IVMRFilterConfig 接口，以参数 VMRMode_Windowless 使用 SetRenderingMode 方法设置完成。最后设置传入的窗口为视频剪辑窗口。

STEP 06 设置捕获图像帧的格式，遍历所有格式是否有预定格式，若没有则以默认格式

捕获。

```
HRESULT CVMR_Capture::InitVideoWindow(HWND hWnd,int width, int height)
{
    HRESULT hr;                                         //返回值
    RECT rcDest;                                        //矩形区域
    IAMStreamConfig *pConfig;                           //流配置接口
    IEnumMediaTypes *pMedia;                            //枚举媒体类型接口
    AM_MEDIA_TYPE *pmt = NULL, *pfnt = NULL;            //媒体类型
    hr = m_pCamOutPin->EnumMediaTypes( &pMedia );  //获取捕获设备的所有媒体类型
    if(SUCCEEDED(hr))
    {
        //把视频的所有格式遍历一遍，看是否有预定的格式
        while(pMedia->Next(1, &pmt, 0) == S_OK)
        {
            if( pmt->formattype == FORMAT_VideoInfo )
            {
              VIDEOINFOHEADER *vih = (VIDEOINFOHEADER *)pmt->pbFormat;
              //当前的格式是否与预定格式相同，即宽和高是否相同
              if( vih->bmiHeader.biWidth == width && vih->bmiHeader.biHeight
              == height )
              {
                   pfnt = pmt;          //记录当前媒体格式
                   break;               //退出循环
              }
              DeleteMediaType( pmt );  //格式不匹配，则删除当前查询的媒体格式
        }
    }
    pMedia->Release();
}
 //获取流配置接口
 hr = m_pCamOutPin->QueryInterface( IID_IAMStreamConfig, (void **) &pConfig );
 if(SUCCEEDED(hr))
 {
  //有预定的媒体格式
    if( pfnt != NULL )
    {
         hr=pConfig->SetFormat( pfnt );
         DeleteMediaType( pfnt );
    }
  //没有预定的格式，读取默认媒体格式
    hr = pConfig->GetFormat( &pfnt );
    if(SUCCEEDED(hr))
    {
       m_nWidth = ((VIDEOINFOHEADER *)pfnt->pbFormat)->bmiHeader.biWidth;
        //读取高
       m_nHeight = ((VIDEOINFOHEADER *)pfnt->pbFormat)->bmiHeader.biHeight;
```

```
            //读取宽
            DeleteMediaType( pfnt );
        }
    }
    //获取传入窗口的区域，以设置显示窗口
    ::GetClientRect (hWnd,&rcDest);
    hr = m_pWC->SetVideoPosition(NULL, &rcDest);  //设置视频窗口位置
    return hr;
}
/* 删除媒体类型*/
void CVMR_Capture::DeleteMediaType(AM_MEDIA_TYPE *pmt)
{
    if (pmt == NULL) {                              //为空则直接返回
        return;
    }
    if (pmt->cbFormat != 0) {                       //格式块大小不为零
        //释放由CoTaskMemAlloc或CoTaskMemRealloc申请的内存块
        CoTaskMemFree((PVOID)pmt->pbFormat);
        pmt->cbFormat = 0;
        pmt->pbFormat = NULL;
    }
    if (pmt->pUnk != NULL) {                        //IUnknown接口不为空
        pmt->pUnk->Release();                       //释放资源
        pmt->pUnk = NULL;
    }

    CoTaskMemFree((PVOID)pmt);
}
```

无论采用 CoTaskMemAlloc 函数还是采用 CreateMediaType 函数分配的内存都可以用这个函数来释放。

STEP 07 从 VMR 中获取一帧图像数据，转换目标颜色空间为 RGB24。m_pFrame 指向 RGB24 格式的数据。

```
/* 从VMR中获取一帧图像*/
DWORD CVMR_Capture::GrabFrame()
{
 if(m_pWC ) {
      BYTE* lpCurrImage = NULL;
         //获取当前的图像数据，实际的数据内容是BMP位图格式
         if(m_pWC->GetCurrentImage(&lpCurrImage) == S_OK)
         {
            //读取图像帧数据lpCurrImage
            LPBITMAPINFOHEADER  pdib = (LPBITMAPINFOHEADER) lpCurrImage;
            //判断图像的宽度和高度是否正确
            if(m_pFrame==NULL||(pdib->biHeight*pdib->biWidth*3)!=m_nFramelen)
            {
```

```
                if(m_pFrame!=NULL)  delete []m_pFrame;      //删除以前申请的内存
                m_nFramelen = pdib->biHeight * pdib->biWidth * 3; //重新申请内存
                m_pFrame = new BYTE [pdib->biHeight * pdib->biWidth * 3] ;
                //申请的内存大小
            }
            if(pdib->biBitCount == 32)
            {
                 DWORD  dwSize=0, dwWritten=0;
                 BYTE *pTemp32;
                 pTemp32=lpCurrImage + sizeof(BITMAPINFOHEADER);
                 //由于采集的是32位RGB，目标要求24位，需要进行转换
                 this->Convert24Image(pTemp32, m_pFrame, pdib->biSizeImage);
            }
            //释放该图像lpCurrImage
            CoTaskMemFree(lpCurrImage);
          }
          else {  return -1;  }
  }else{  return -1;  }
         return m_nFramelen;
}
```

上述程序实现获取当前显示图像，如果格式不正确则重新申请空间。由于采集的图像多数为 32 位，所以需要转换成 24 位以便于算法直接处理，捕获的图像帧放置在类 CVMR_Capture 的成员变量 m_pFrame 中，该函数返回图像帧的数据大小。

STEP 08 颜色空间转换 ARGB32 到 RGB24。

```
/* ARGB32 to RGB24 */
bool CVMR_Capture::Convert24Image(BYTE *p32Img, BYTE *p24Img,DWORD dwSize32)
{
 if(p32Img != NULL && p24Img != NULL && dwSize32>0)       //确认指针合法
 {
        DWORD dwSize24;
        dwSize24=(dwSize32 * 3)/4;         //RGB32与RGB24的像素点空间只差了一个字节
        BYTE *pTemp=p32Img,*ptr=p24Img;
        for (DWORD index = 0; index < dwSize32/4 ; index++)
        {
             unsigned char r = *(pTemp++);   unsigned char g = *(pTemp++);
             unsigned char b = *(pTemp++);   (pTemp++);   //跳过alpha分量
             *(ptr++) = b;   *(ptr++) = g;   *(ptr++) = r; 记录RGB 3分量
        }
 } else {
        return false; //指针不合法
 }
 return true;
}
```

RGB32 与 RGB24 格式只差一个分量 alpha，所以转换时把每个像素点的该分量丢掉，即

可得到 RGB24 格式。另外类 CVMR_Capture 还有其他常见成员函数，如保存滤波器链表。

9.3.4 界面设计

在这个实例中我们要实现的任务是在预览视频的同时从 VMR 中定时捕捉图像数据，然后从 ARGB32 转换到 YUV420（I420）格式，并保存到*.yuv 文件。预览可暂停、重启。能够在不退出应用程序的情况下，终止捕捉图像。

应用 VC++ 2005 应用程序向导建立对话框程序框架，项目名称为 9-2。建立过程请参考 8.1.3 章节。

添加控件：6 个 Button、1 个 Combo Box、1 个 Picture Control，如表 9-2 所示。根据其功能修改所有控件的 ID。

表 9-2　控件功能列表

方法	说明
ID_PREVIEW	预览视频
ID_PAUSEPLAY	暂停/开始预览
ID_CAPTURE	捕捉图像
ID_STOPCAP	终止捕捉图像
ID_SAVEGRAPH	保存滤波器链表
ID_EXITAPP	退出程序（终止预览捕捉）
IDC_LISTDEVICE	设备列表组合框
IDC_VIDEO_WINDOW	显示捕捉的视频图像

修改控件大小，重排其位置。添加控件后的主界面设计如图 9-19 所示。

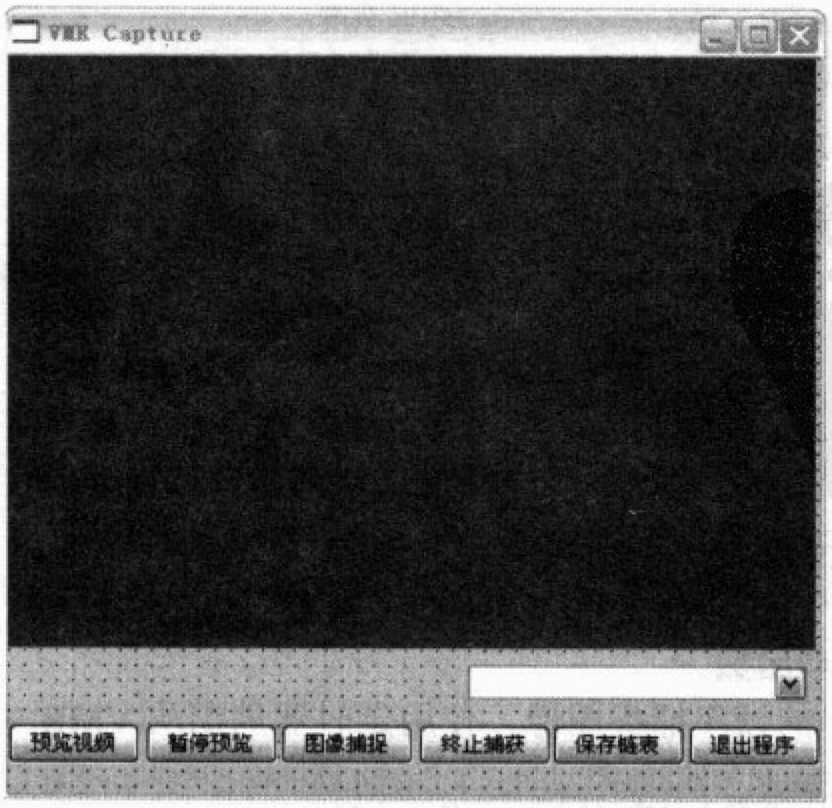


图 9-19　VMR Capture 主界面设计

控件与变量捆绑。为便于控制，把 IDC_VIDEO_WINDOW 控件、IDC_LISTDEVICE 控件分别与变量捆绑，途径是选中控件的右键菜单中的“添加变量”命令，激活“添加成员变量向导”。添加后的代码如下。

```
// 显示捕获的图像
 CStatic m_videoWindow;
// 组合框列表，显示设备名称
CComboBox m_listCtrl;
```

添加功能按钮在线提示 ToolTip，添加过程请参照 9.2.3 的（5）步骤。功能按钮的消息响应，即单击按钮的事件处理。双击某按钮，实现单击按钮事件处理函数的添加。为了使用类 CVMR_Capture 的变量和函数，在应用类中 CMy92Dlg 引入并定义。

```
CVMR_Capture m_VMRCap;
```

在类 CMy92Dlg 中定义成员变量。

```
CToolTipCtrl m_tooltip;                //在线提示控件
CVMR_Capture m_VMRCap;                 //VMR类
ColorSpaceConversions conv;            //颜色空间转换类
UINT m_timerID;                        //定时器ID
CString m_yuvFileName;                 //YUV文件名
CFile m_pFile;                         //文件类
BOOL m_fileState;                      //文件状态：打开=true ,关闭=false
int m_imageWidth;                      //图像宽度
int m_imageHeight;                     //图像高度
BYTE *p_yuv420;                        //YUV420空间
```

定义宏以为修改提供方便。

```
#define TIMER_ID 1                     //定时器ID
#define TIMER_DELAY 20                 //定时器周期
#define CAMERA_WIDTH  352              //图像宽度
#define CAMERA_HEIGHT  288             //图像高度
```

对话框初始化，初始化变量。

```
this->m_VMRCap.EnumDevices(this->m_listCtrl);
                                         //枚举系统采集设备，设备名称添加到组合框
 this->m_listCtrl.SetCurSel(0);          //选择0索引的设备
 m_yuvFileName = _T("");                 //yuv文件名
 m_fileState = FALSE;                    //文件当前状态
 m_imageWidth = CAMERA_WIDTH;            //预采集的图像宽度
 m_imageHeight= CAMERA_HEIGHT;           //预采集的图像高度
 p_yuv420 = NULL;                        //YUV420数据空间
```

添加定时器消息处理函数

为了以不低于 25f/s 的帧率捕捉图像，使用定时器周期从 VMR 中读取图像数据。

```
void CMy92Dlg::OnTimer(UINT_PTR nIDEvent)
```

```
{
 //TODO：在此添加消息处理程序代码和/或调用默认值
 if (nIDEvent==TIMER_ID)                                   //定时器ID
 {       //定时获取图像帧，转换颜色空间，写文件
         if (m_fileState)                                  //文件已经打开
         {
                 DWORD dwSize;
                 dwSize = this->m_VMRCap.GrabFrame();      //获取图像帧数据
                 if (dwSize >0 )
                 {
                         BYTE *pImage;
                         this->m_VMRCap.GetFrame(&pImage); //读取图像内容
                         //把RGB24转换为YUV420
                         conv.RGB24_to_YV12(pImage,this->p_yuv420,m_imageWidth,
                         m_imageHeight);
                         //写YUV420数据到文件
                         m_pFile.Write(p_yuv420,m_imageWidth*m_imageHeight*3/2);
                 }
         }
 }
 CDialog::OnTimer(nIDEvent);
}
```

成员函数 GrabFrame 是真正地获取图像数据，而 GetFrame 只是获取前者保存的数据指针。为便于大部分的算法处理，进行了颜色空间转换 RGB24 到 YUV420。该转换功能使用了经典的 cscc.lib 库。

添加功能按钮的单击消息处理函数。双击“预览视频”按钮，添加事件处理代码：

```
void CMy92Dlg::OnBnClickedPreview()
{
 //TODO：在此添加控件通知处理程序代码
 HWND hwnd = this->m_videoWindow.GetSafeHwnd();            //获取显示窗口句柄
 int id = this->m_listCtrl.GetCurSel();                    //获取当前设备索引
 HRESULT hr = m_VMRCap.Init(id,hwnd,m_imageWidth,m_imageHeight);
                                                           //创建预览的所有任务
 if (FAILED(hr))
       AfxMessageBox(_T("无法创建滤波器链表！"));          //创建失败
}
```

上述程序实现视频预览功能，首先获取显示窗口的句柄，根据设定的设备 ID，图像宽度和高度创建预览的所有任务，并开始预览。创建过程任何一步有错误都认为创建失败。视频预览效果如图 9-20 所示，当画面有运动时，图像明显比使用经典采集技术方法要流畅。

图 9-20　VMR 视频预览

双击“暂停预览”按钮，添加事件处理代码：

```
void CMy92Dlg::OnBnClickedPauseplay()
{
 //TODO: 在此添加控件通知处理程序代码
 static int c=0;
 if (!c) {
        GetDlgItem(ID_PAUSEPLAY)->SetWindowTextW(_T("开始预览")); //改变按钮标题
        c = 1;
 } else {
        GetDlgItem(ID_PAUSEPLAY)->SetWindowTextW(_T("暂停预览")); //改变按钮标题
        c = 0;
 }
 m_VMRCap.Pause();                                                //暂停预览
}
```

上述程序使用 CVMR_Capture 类的成员函数 Pause 实现暂停功能，按钮标题根据当前功能做更改，暂停效果如图 9-21 所示。

图 9-21　暂停预览效果图

双击“图像捕捉”按钮，添加事件处理代码：

```
void CMy92Dlg::OnBnClickedCapture()
{
 //TODO: 在此添加控件通知处理程序代码
 CFileDialog dlg(FALSE);                              //另存为文件对话框
 if (dlg.DoModal()==IDOK)
 {
       m_yuvFileName = dlg.GetPathName();             //获取输入的文件路径名
       BOOL ret = m_pFile.Open(m_yuvFileName,         //打开、创建文件
                              CFile::modeCreate|CFile::modeWrite|
                               CFile:: typeBinary);
       if (!ret) {
            AfxMessageBox(_T("创建YUV文件失败！"));
            return ;
       } else m_fileState = ret;
       if (!p_yuv420)                                 //分配YUV420图像空间
            p_yuv420 = new BYTE [m_imageWidth*m_imageHeight*3/2];
       this->KillTimer(TIMER_ID);                     //清除以前创建的定时器
       this->SetTimer(TIMER_ID,TIMER_DELAY,NULL); //启动定时器
 }
}
```

上述程序完成 YUV420 文件的获取与创建，然后创建 YUV420 空间，启动采集图像的定时器，采集间隔为 TIMER_DELAY。该消息处理函数的效果如图 9-22、9-23 所示。注意在捕获图像前，需要先预览视频，否则程序会报错。

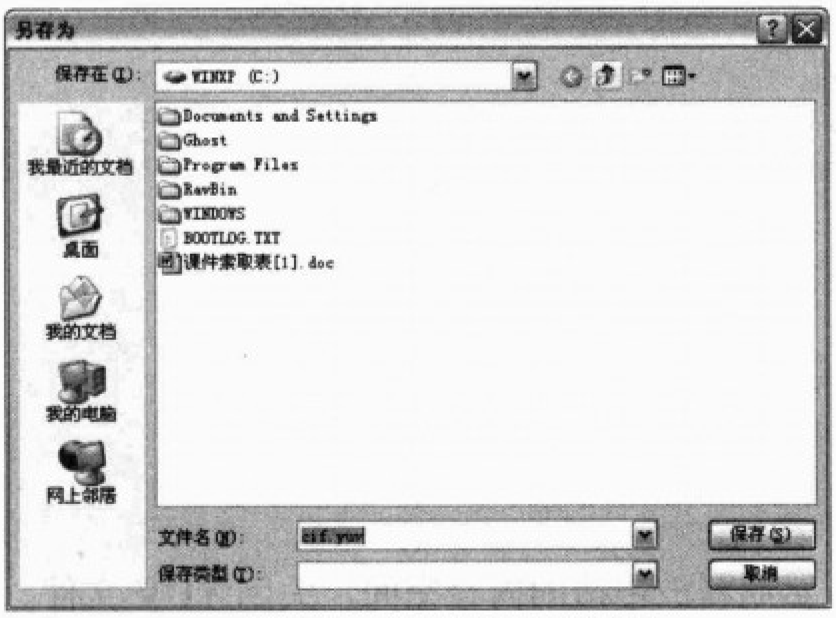


图 9-22 “另存为”对话框

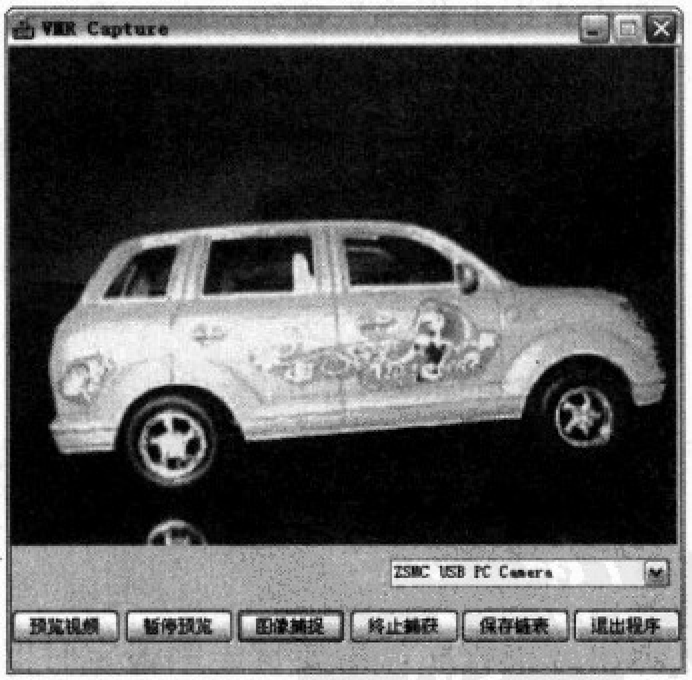


图 9-23 同时预览和捕获的主界面

双击“终止捕捉”按钮，添加事件处理代码：

```
void CMy92Dlg::OnBnClickedStopcap()
{
 //TODO: 在此添加控件通知处理程序代码
 KillTimer(TIMER_ID);
 if (p_yuv420) {
```

```
        delete [] p_yuv420;                           //释放YUV420空间
        p_yuv420 = NULL;
  }
  if (m_fileState) {
        m_pFile.Close();                              //关闭YUV文件
        m_fileState = FALSE;
  }
}
```

上述程序实现终止捕获存储任务，关闭当前的 YUV 文件，图像预览任务继续。

双击“保存图表”按钮，添加事件处理代码：

```
void CMy92Dlg::OnBnClickedSavegraph()
{
 //TODO: 在此添加控件通知处理程序代码
 HRESULT hr;
 CFileDialog dlg(FALSE);                             //另存为对话框
 if (dlg.DoModal()==IDOK)
 {
         CString inFileName = dlg.GetPathName();  //获取输入的文件路径名
         m_VMRCap.SaveGraph(inFileName);          //保存graph到文件中
 }
}
```

保存滤波器链表到文件中，便于使用 GraphEdit 程序运行，掌握应用程序系统中滤波器的使用情况。

双击“退出程序”按钮，添加事件处理代码。程序隐含调用了类 CVMR_Capture 的析构函数，释放资源和 COM 库。

```
void CMy92Dlg::OnBnClickedExitapp()
{
 //TODO: 在此添加控件通知处理程序代码
 KillTimer(TIMER_ID);
 if (p_yuv420)
        delete [] p_yuv420;
 if (m_fileState)
        m_pFile.Close();
 CDialog::OnOK();
}
```

退出应用程序，需要首先销毁定时器，然后释放申请的内存空间，关闭 YUV 文件。

9.4 系统效果演示

前面介绍了使用 DirectShow 经典视频采集技术和使用 VMR 技术实现视频采集及这两个案例的设计过程，展现了 DirectShow 的强大功能。下面是这两个实例系统的效果展示。

9.4.1 实例一 系统效果演示

实例一 CaptureVideo 使用了 DirectShow 的经典视频采集技术，操作简单、系统功能较强，主界面如图 9-24 所示。

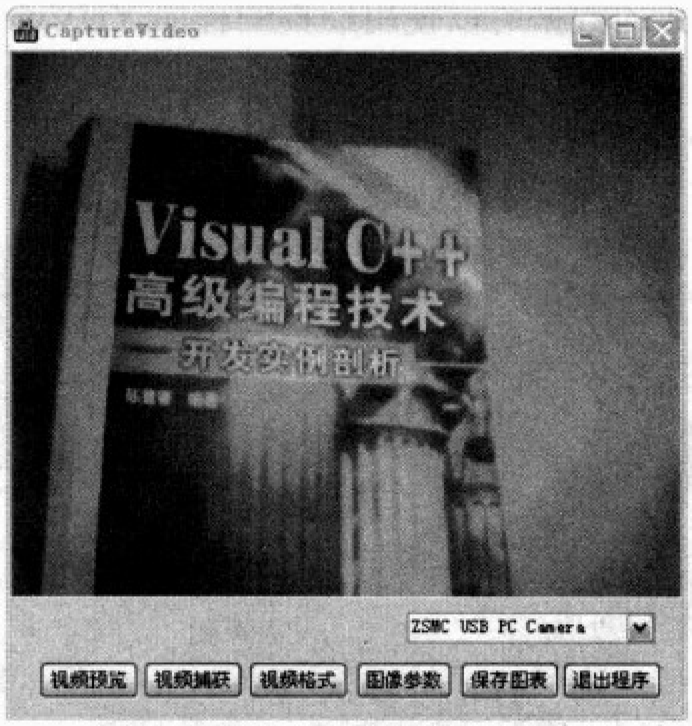


图 9-24 经典视频采集技术实例

STEP 01 先“视频预览”，后“视频捕获”。视频捕获文件名设置为“*.avi”。

STEP 02 “视频格式”配置视频的分辨率和帧率，如图 9-25 所示。“图像参数”配置视频图像的参数，如图 9-26 所示。

STEP 03 “保存图表”命名为“*.grf”文件，供 GraphEdit 分析 DirectShow 滤波器工作原理。

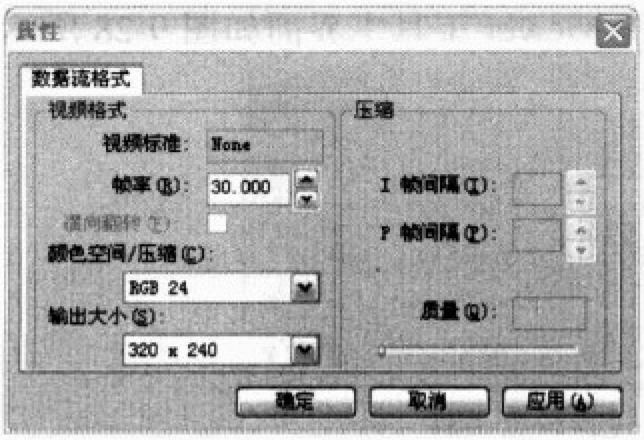


图 9-25 视频采集设备视频格式配置

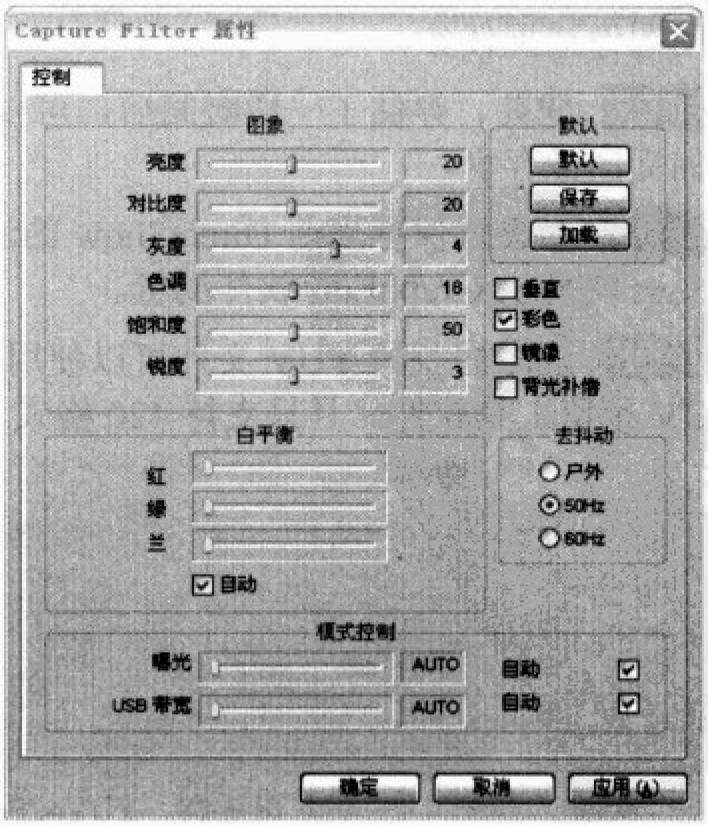


图 9-26 视频采集设备图像参数配置

9.4.2 实例二 系统效果演示

实例二使用了无窗口模式的 VMR-9 技术，流畅、清晰预览视频，实时稳定捕捉图像帧数据。从 ARGB32 转换到 RGB24，再转换到视频编码算法常用的 YUV420（I420）格式，最后写文件保存。YUV420 格式的数据可以使用经典的程序 YUVviewer 浏览观看。使用本实例的一般步骤如下。

STEP 01 单击“预览视频”按钮，开始 VMR 视频捕获预览，该实例支持暂停预览功能，单击“暂停预览”按钮实现该功能，效果如图 9-27 所示。

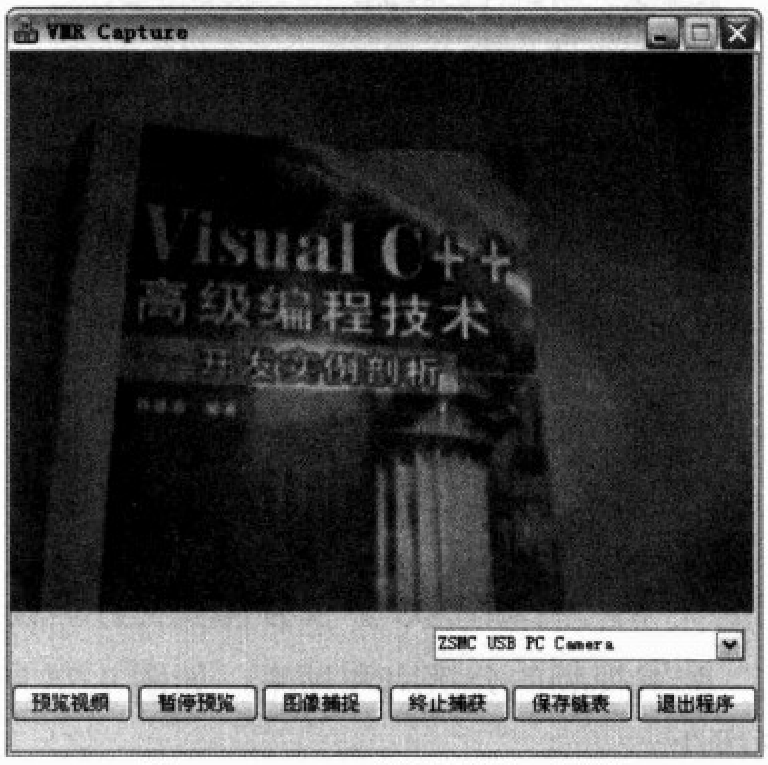


图 9-27 VMR 视频捕获、预览

STEP 02 在确定已经看到捕获的图像后单击“图像捕捉”按钮，设置保存的 YUV 文件名，启动图像捕捉任务。

STEP 03 捕获、存储了一段时间后，可以单击“终止捕获”按钮，视频预览任务继续，但不再捕捉图像存盘。

STEP 04 如果想保存当前 DirectShow 的滤波器工作链表，可以单击“保存链表”按钮，保存的*.grf 文件供 GraphEdit 软件浏览。

STEP 05 程序捕获的 YUV 文件可以使用 YUVviewer 应用程序来浏览，该程序是一款经典的 YUV 预览工具软件，且只支持 420 格式，YUVviewer 工具主界面如图 9-28 所示。

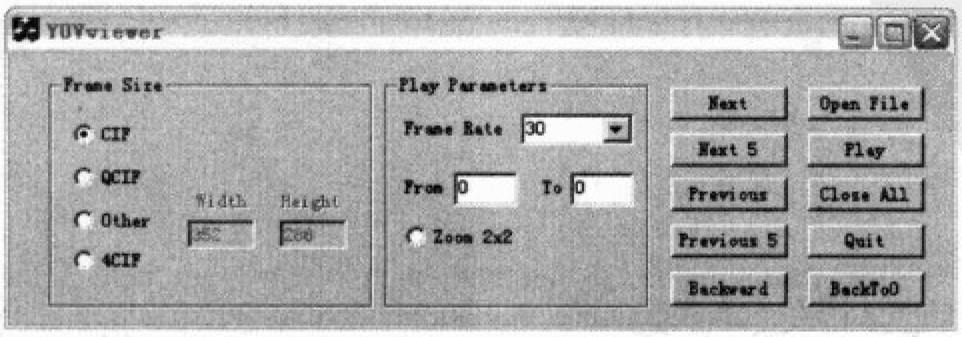


图 9-28 YUV420/I420 视频浏览工具

9.5 本章小结

视频预览和捕捉存盘同时进行，是视频捕获的基本要求。案例介绍了视频采集原理：创建滤波器链表、添加捕获设备滤波器、控制媒体数据的播放/暂停等操作，同时详细地展示了设计主界面的技术过程。DirectShow SDK 提供了经典的视频采集技术 ICaptureGraphBuilder2 接口，利用该接口的所属方法 RenderStream 自动建立、链接滤波器链表，快速构建视频采集程序框架，完成基本的视频预览和捕捉存盘，特点是数据对用户来说是透明的。VMR 技术实现视频图像捕获可以提供图像流畅显示和图像数据的直接访问，便于用户算法直接处理，捕获的数据格式是 ARGB32，在实例中实现了 ARGB32 到 RGB24，再到 YUV420 格式的转换。实例二特点是用户直接访问图像数据，图像显示更流畅、更清晰。

第 10 章

VFW 技术实现视频处理通用平台

VFW（Video for Windows）是视频开发应用的一种早期技术。数字视频捕获是一项极其复杂的工作。Microsoft 的 Visual C++从 4.0 版开始就支持 Video for Windows（简称 VFW）了，这给视频捕获编程带来了很大的方便。虽然现在已经有功能更为强大的 DirectShow SDK，但是 VFW 以其结构清晰、技术简单等特点，对于已经在使用 VC++ 2005 开发视频应用的用户来说，仍然是宠爱有加、情有独钟。

本章在介绍、分析 VFW 的开发技术流程的基础上，实现视频的实时捕获预览。通过两个平台应用“使用系统插件实现视频编/解码”和“使用 XviD CODEC 算法实现视频编/解码”来展示 VFW 技术的视频处理通用平台。用户可以直接拿来应用作为开发平台，也可以继续丰富该平台案例。开发或使用本案例前，首先确定系统已经安装了 USB 摄像头。

案例中的 2 个平台应用都是基于 VC++ 2005 的单文档应用程序框架，平台的功能控制通过标准菜单实现，视频预览窗口可任意改变大小和移动位置。由于显示使用了 DrawDibxxx 函数，在任意改变窗口大小时，图像质量显示高、占用 CPU 资源少。平台应用二完全可以作为开发者使用自己的算法实时处理视频、显示处理结果的案例。

本章重点

- VFW 开发技术流程分析
- 使用 VFW 实现视频捕获和预览
- 使用系统插件实现视频编/解码
- 使用 XviD CODEC 算法实现视频编/解码
- 平台应用效果展示

PDG

10.1 VFW开发技术流程分析

VFW 技术可以实现视频的捕获、压缩和回放等功能，可以设置视频的格式，如分辨率、帧率等。使用 VFW 进行视频开发有个一般的开发流程即打开、使用和关闭 VFW 设备。

10.1.1 技术概述

VFW 的视频捕获主要由 AVICap 窗口类来完成。AVICap 窗口类为应用程序提供了一个基于消息的接口，使应用程序不仅能访问视频和波形音频硬件，还能在将视频流采集到硬盘的过程中进行控制。AVICap 支持实时的视频流采集和单帧采集并提供对视频源的控制，能直接访问视频缓冲区，不需要分配中间文件，实时性强、效率高，并且通过简单的函数调用，可以很方便地设置在系统中已经注册的压缩和解压缩驱动，实现视频流、音频流的实时压缩。

概括来讲，AVIcap 窗口类的主要功能有如下几个方面。

- 动态连接和断开音/视频采集设备。
- 设置视频采集速率。
- 设置视频格式及来源。
- 指定保存数据的文件名及路径。
- 提供 Overlay（视频叠加）和 Preview（视频预览）两种显示模式。
- 实时采集和存储单帧视频图像，采集视/音频流并存入 AVI 影音文件。
- 通过 AVICap 的回调函数（宏）获得有关视频采集的状态和数据。

这些功能的实现是通过调用宏、消息和函数来完成的，开发人员在自己的应用程序中可以根据具体的应用，灵活运用宏（cap_xx）、消息和函数来实现这些功能。使用 VFW 进行视频采集的基本流程如下。

STEP 01 创建视频采集窗口，这是所有采集工作及 VFW 设置的基础。

STEP 02 注册系统回调函数，使得在采集过程中 VFW 自动回调相应的函数。

STEP 03 连接视频采集设备，获取视频采集设备的能力及状态信息。

STEP 04 获取和设置采集窗口参数，如视频分辨率、颜色空间和采集速率等。

STEP 05 设置采集窗口的显示模式为 Preview 或 Overlay。

STEP 06 根据不同需要选择采集图像到缓存还是文件。

STEP 07 视频采集结束时，将采集窗口与驱动断开连接，否则将导致视频驱动无法释放，其他程序将不能使用该采集设备。

10.1.2 VFW采集开发流程图

视频采集的基本流程如图 10-1 所示，VFW 的使用开发都可以通过调用以 cap 开头的宏来实现。

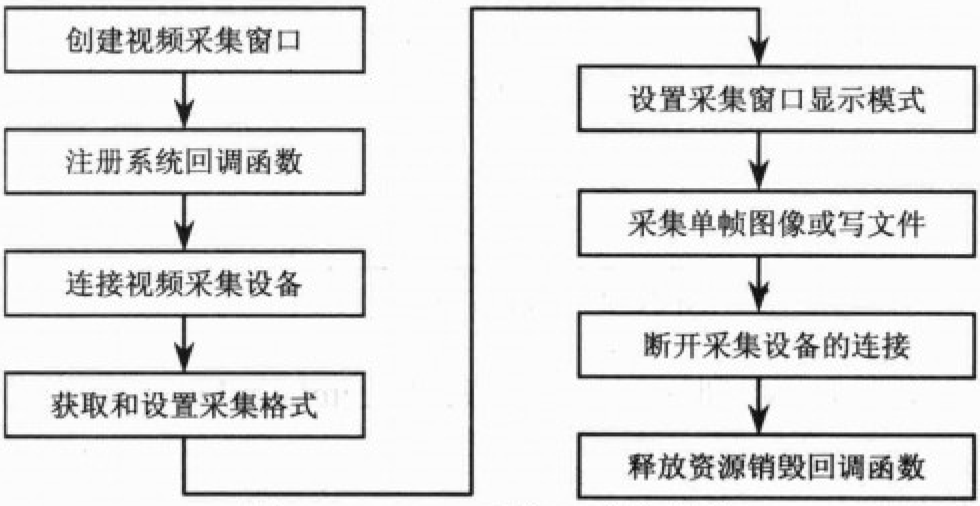


图 10-1 VFW 采集开发流程图

10.2 使用VFW实现视频捕获和预览

使用 VFW 的宏函数实现视频捕获和预览，应用程序简单、控制灵活。在前面的案例中，我们都是基于 VC 2005 的对话框应用程序框架实现编程开发，其实很大一部分的应用程序都是基于此类的。而另一部分应用就是基于文档的编程开发，单文档（SDI）是主流。文档应用程序的最大特点是有标准菜单、客户区域任意调整；缺点是相比对话框类，开发难度加大。

10.2.1 建立单文档应用程序

现在基于 VC++ 2005 的项目建立向导创建一个单文档、无工具栏和状态栏的应用程序。

STEP 01 启动 VC++ 2005，选择“文件”→“新建”→“项目”命令。在“新建项目”对话框中，选择“Visual C++”项目类型下的“MFC”选项，选中“模板”的“MFC 应用程序”，命名为 vfwapp。单击“确定”按钮开启“MFC 应用程序向导”。根据向导提示逐步建立应用程序。

STEP 02 为平台应用一添加功能菜单“VFW 采集”，并插入 4 个子项，ID 设置如表 10-1 所示，“VFW 采集”菜单如图 10-2 所示。

表 10-1 VFW 采集子菜单

标题	ID 号	备注
初始化 VFW	ID_VFW_INITVFW	创建窗口、连接设备及注册回调函数
配置视频格式	ID_VFW_VIDEOFORMAT	预览前配置视频格式

（续表）

标题	ID 号	备注
预览视频	ID_VFW_PREVIEWVIDEO	开始捕获、预览视频
捕获编/解码	ID_VFW_CODEC	对捕获的视频做编码和解码处理

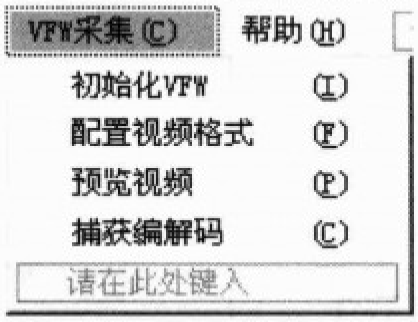


图 10-2 “VFW 采集”菜单

STEP 03 修改窗口样式，在创建窗口函数 PreCreateWindow 中添加以下代码。

```
cs.cx = 352+8;
cs.cy = 288+54;
cs.lpszName =_T("VFW采集");          //要设置的标题
cs.style&   =~FWS_ADDTOTITLE;       //覆盖应用程序名称，只显示lpszName
```

STEP 04 因为 VFW 的函数库封装在了 vfw32.lib 中，如果对要基于 VFW 进行开发，则需要引入该库及 vfw.h 头文件。我们的 VFW 编程是在主窗口 CmainFrame 中，所以在 CmainFrame.h 中引入头文件。另外，视频图像显示使用了 DibDraw_xx 函数，所以在 CvfwappView 中也需要引入头文件。

```
#include <vfw.h>
```

引入库 vfw32.lib 有两种方法。

- 直接在源程序中引入。

```
#pragma comment(lib,"vfw32.lib");
```

- 在 VC2005 开发环境项目“属性”中的“配置属性”的“链接器”中的“附加依赖项”输入“vfw32.lib”，如图 10-3 所示。

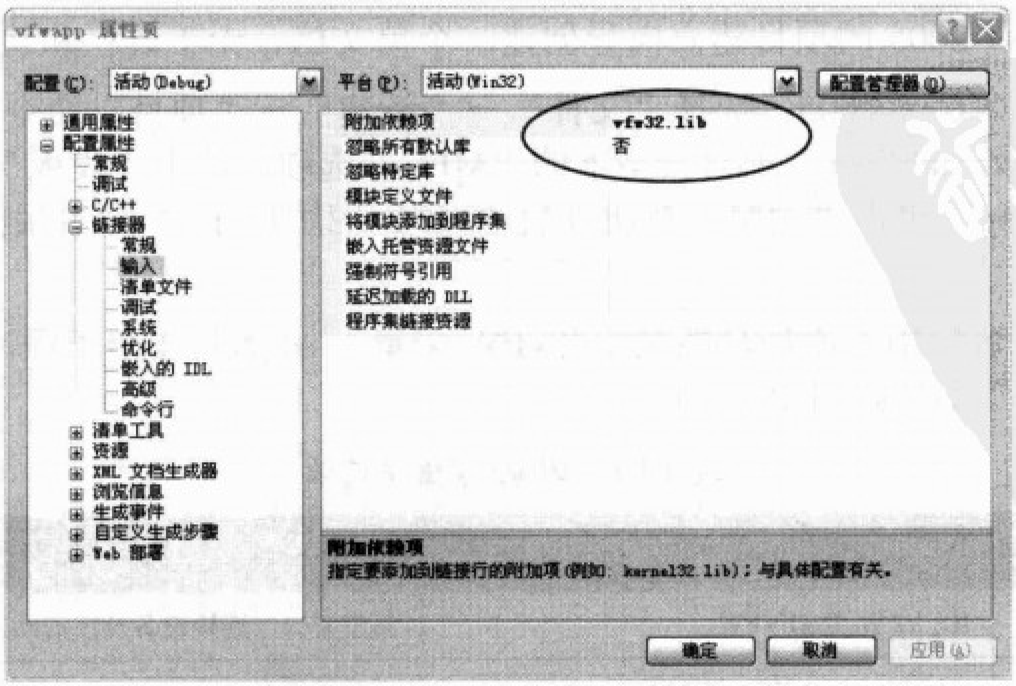


图 10-3 项目属性引入附加库

STEP 05 在使用 VFW 开发时，我们定义了一些全局变量以控制采集、预览等功能。

```
#define XDIM 704                              //图像宽度，预设的足够大
#define YDIM 576                              //图像高度，预设的足够大
unsigned char bufo[XDIM*YDIM*3+40];           //送客户区显示的图像，RGB格式
//视频压缩管理器（VCM）使用的参数
HIC             hic1;                         //编码器句柄
HIC             hic2;                         //解码器句柄
LPBITMAPINFO lpbiIn;                          //输入格式
LPBITMAPINFO lpbiTmp;                         //编码器输出格式
LPBITMAPINFO lpbiOut;                         //输出格式
COMPVARS        pc;                           //编码设置结构体
BOOL            IsKeyFrame;                   //关键帧标志
BOOL            bSaveAVI=FALSE;               //保存为AVI
Long            FrameSize;                    //编码后码流大小
Int             CurrentID;
BOOL            bPreview=FALSE;               //预览标志
int             nStreamLength;                //视频流长度
int             nOstreamSize;                 //输出视频流的大小
CFrameWnd       m_wndSource;                  //创建的新窗口
HWND            m_hWndCap;                    //VFW设备窗口
CRect       disRect;                          //显示窗口的客户区域
CMainFrame      *pMainFrame=NULL;             //MainFrame 指针
enum            VFW_STATE{PREVIEW,ENCDEC};
VFW_STATE       m_vfwState = PREVIEW;
```

10.2.2 创建视频窗口

VFW 的设备控制和视频数据的读写均通过窗口句柄来实现，所以首先需要创建视频窗口，在源程序中使用函数 capCreateCaptureWindow 创建视频窗口。另外需要注意的是期望采集的视频清晰、流畅地在主窗口的客户区显示，但是如果把该主窗口句柄与 VFW 函数 capCreateCaptureWindow 的参数视频捕捉窗口捆绑的话，则显示区域边缘不会被刷新。所以可以重新创建一个窗口，但是不显示，而 VFW 采集的数据是显示在主窗口的客户区的，程序实现如下。

```
CFrameWnd    m_wndSource;                      //创建的新窗口
HWND m_hWndCap;                                //VFW视频捕获窗口句柄
if(!m_wndSource.CreateEx(WS_EX_TOPMOST,        //置顶显示
                        NULL,
                        _T("Source"),WS_OVERLAPPED|WS_CAPTION,
                        CRect(0,0,352,288),NULL,0))
    return;
    m_hWndCap = capCreateCaptureWindow(_T("Capture Window"),  //视频捕捉窗口名称
                        WS_CHILD|WS_VISIBLE,    //视频捕捉窗口样式及风格
```

```
                    0,0,352,288,                    //视频捕捉窗口坐标及视频的宽高
                    m_wndSource.m_hWnd,0);          //视频捕捉窗口父窗口句柄
```

上述程序首先创建一个窗口，但是不显示。接着创建 VFW 视频捕获窗口句柄 m_hWndCap。VFW 的所有控制都是通过该句柄来完成的，创建的窗口的参数见上述代码的说明或参考 MSDN。

10.2.3 设计回调函数

VFW 的很多功能都是通过回调函数来实现的，如读取视频图像、获取设备当前状态等。使用回调函数的过程是在系统中先注册回调函数，然后 VFW 就根据设备状态自动调用回调函数，常用的注册回调函数如表 10-2 所示。

表 10-2 VFW 注册回调函数

注册回调函数名称	注册函数作用
CapSetCallbackOnCapControl	用于控制捕获的开始和结束
CapSetCallbackOnError	在出错的时候调用
CapSetCallbackOnFrame	在预览图像帧被捕获了的时候调用
CapSetCallbackOnStatus	在状态（status）改变的时候调用
CapSetCallbackOnVideoStream	在流捕获期间，当一个新的视频缓冲区可用时调用
CapSetCallbackOnWaveStream	在流捕获期间，当一个新的音频缓冲区可用时调用
CapSetCallbackOnYield	产生一个后台线程进行视频捕获处理

这些注册回调函数的参数形式均为：

```
capSetCallBackOn_xx(HWND hwnd, CAP_XX_CALLBACK fpProc);
```

hwnd 为视频采集窗口的句柄，fpProc 为回调函数指针。

一般用到的回调函数有 3 个：出错回调、状态改变回调和视频获取回调。

当采集、预览等出错时回调该函数。

```
    LRESULT CALLBACK EXPORT ErrorCallbackProc(HWND hWnd, int nErrID, LPSTR
lpErrorText)
    {
     if(nErrID==0)    //错误ID不为0
         return TRUE;
     MessageBox(NULL,(LPCWSTR)lpErrorText,_T(""),MB_OK);
     return TRUE;
    }
```

上述程序实现如果 VFW 出错，则给出提示信息。

读取 VFW 状态时的回调函数如下。

```
LRESULT FAR PASCAL StatusCallbackProc(HWND hWnd, int nID, LPSTR lpStatusText)
{
 if(nID==IDS_CAP_END){ //停止捕捉
        //音频和视频或者仅仅视频
       if((CurrentID==IDS_CAP_STAT_VIDEOAUDIO)||(CurrentID==IDS_CAP_STAT_
VIDEOONLY))
              return TRUE;
 }
 CurrentID = nID;
 return (LRESULT) TRUE;
}
```

视频缓冲区填充满时，回调该函数。我们对视频数据的处理均在此函数实现，现在我们要求把捕获的数据复制到显示缓冲区并显示出来。

```
LRESULT FAR PASCAL VideoCallbackProc(HWND hWnd, LPVIDEOHDR lpVHdr)
{
 unsigned char *bufi, *buf;
 bufi = new unsigned char[lpVHdr->dwBytesUsed+40];    //采集的原视频数据
 buf = new unsigned char[lpVHdr->dwBytesUsed];        //编码后的压缩流
 memcpy((void *)(bufi), lpVHdr->lpData, lpVHdr->dwBytesUsed);
 unsigned char *buf1;           //临时指针
 buf1 = buf;                    //获取指针
 {
       memcpy(&bufo[40],bufi,lpVHdr->dwBytesUsed);        //获取视频数据，bufo的
                                                            前40字节为BMP头
 }
 //发送消息给视图View，更新客户区
 pMainFrame->GetActiveView()->InvalidateRect(NULL,FALSE);
 delete bufi; //释放内存
 delete buf;  //释放内存
 return (LRESULT) TRUE;
}
```

上述程序是 VFW 工作的最主要的函数，在这里获取捕获的视频数据，进行数据的各种处理。采集到的视频数据放置在 bufo 空间中，前 40 字节是 BMP 位图的头信息。

注册回调函数的代码如下。

```
capSetCallbackOnError(m_hWndCap,(FARPROC)ErrorCallbackProc);              //出错回调
 capSetCallbackOnStatus(m_hWndCap,(FARPROC)StatusCallbackProc);           //状态回调
 capSetCallbackOnVideoStream(m_hWndCap,(FARPROC)VideoCallbackProc);   //读取视频流
```

在实现了回调函数后，使用 VFW 的标准注册回调函数注册用户自己的回调函数。参数 m_hWndCap 为 VFW 视频窗口句柄，ErrorCallbackProc、StatusCallbackProc、VideoCallbackProc 为注册回调函数的指针。

10.2.4 视频图像显示设置

在视频图像的显示中，我们采用 VFW 的 DrawDib_xx 函数，窗口任意比例、实时地显示图像。

STEP 01 在 vfwappView.h 中，引入 vfw.h，即#include <vfw.h>。

STEP 02 在类中定义成员变量 HDRAWDIB m_hdd。

STEP 03 在 vfwappView.cpp 中打开、关闭 DrawDib_xx 功能。

```
CvfwappView::CvfwappView()
{
 //TODO: 在此处添加构造代码
      m_hdd=DrawDibOpen();    //打开DrawDib_xx
}
CvfwappView::~CvfwappView()
{
     DrawDibClose(m_hdd);     //关闭DrawDib_xx
}
```

使用 VFW 的 DrawDib_xx 函数，首先在类构造函数中打开 Dib 设备以获得 Dib 句柄。在类析构函数中关闭 Dib 设备。

STEP 04 最后显示图像，实现代码如下。

```
void CvfwappView::OnDraw(CDC * pDC)
{
 CvfwappDoc* pDoc = GetDocument();
 ASSERT_VALID(pDoc);
 if (!pDoc)
      return;
 //TODO: 在此处为本机数据添加绘制代码
 RECT rect;
 GetClientRect(&rect);  //获得矩形客户区域
/*****************设置位图信息*******************************************/
 ((LPBITMAPINFOHEADER)bufo)->biSize = sizeof(BITMAPINFOHEADER);    //结构体大小
 ((LPBITMAPINFOHEADER)bufo)->biWidth = lpbiIn->bmiHeader.biWidth;    //图像宽度
 ((LPBITMAPINFOHEADER)bufo)->biHeight = lpbiIn->bmiHeader.biHeight; //图像高度
 ((LPBITMAPINFOHEADER)bufo)->biPlanes = 1;                           //平面
 ((LPBITMAPINFOHEADER)bufo)->biBitCount = 24;                        //24位位图
 ((LPBITMAPINFOHEADER)bufo)->biCompression = BI_RGB;                 //RGB格式
//图像数据大小
 ((LPBITMAPINFOHEADER)bufo)->biSizeImage =
                   lpbiIn->bmiHeader.biWidth*lpbiIn->bmiHeader.biHeight*3;
 ((LPBITMAPINFOHEADER)bufo)->biClrUsed = 0;          //位图实际使用的颜色表中的颜色数
 ((LPBITMAPINFOHEADER)bufo)->biXPelsPerMeter = 0; //为位图指定目标设备的水平分辨率
 ((LPBITMAPINFOHEADER)bufo)->biYPelsPerMeter = 0; //为位图指定目标设备的垂直分辨率
```

```
((LPBITMAPINFOHEADER)bufo)->biClrImportant = 0;  //重要颜色的数目
/*****************设置位图信息*******************************************/
DrawDibDraw(m_hdd,
            pDC->GetSafeHdc(),
            rect.left, rect.top,    rect.right, rect.bottom,
            (LPBITMAPINFOHEADER)bufo, &bufo[40],    //bufo的前40个字节是BMP头
            0,0, -1,-1,
            0);
}
```

上述程序实现显示采集的 bfuo 中的图像数据，由于显示的位图，所以在显示前需要设置位图的格式。显示函数 DrawDibDraw 的特点是可以不关心目标客户区的大小，函数会自动填充满客户区。这种功能对于全屏显示特别有用，因为把实际图像插值放大成全屏大小是非常占用系统资源的。该函数具有显示图像速度快、占用系统资源低、图像流畅等特点。

10.2.5 捕获预览视频

我们在捕获视频时总希望先预览视频，然后再进行捕获处理。根据 10.2.2 节的视频采集开发流程，我们编程实现视频的捕获预览。注册完回调函数，连接视频设备。这里我们假定系统只安装了一个视频采集设备。

连接视频设备，实现代码如下。

```
CAPDRIVERCAPS m_caps;
 //连接视频设备
 capDriverConnect(m_hWndCap,0);  //(HWND m_hWndCap, int index);
 //获取驱动器的性能参数
 capDriverGetCaps(m_hWndCap,&m_caps,sizeof(CAPDRIVERCAPS));
 if (m_caps.fHasOverlay)                          //设备是否支持overlay
     capOverlay(m_hWndCap,TRUE);
 //设置预览速率开始预览
 capPreviewRate(m_hWndCap,1000/25);               //设置预览频率
 capPreview(m_hWndCap,bPreview);                  //开始预览
//获取视频格式
 fsize = capGetVideoFormatSize(m_hWndCap);        //获取视频格式大小
 capGetVideoFormat(m_hWndCap, lpbiIn, fsize);     //读取fsize大小的视频数据格式
 AfxMessageBox(_T("初始化成功! "));                //初始化成功
```

连接 VFW 视频设备时，首先连接 VFW 视频设备的驱动，然后获取驱动器的性能参数以判断设备是否支持 overlay 模式，设置预览速率和视频格式。

捕获预览视频，实现代码如下。

```
void CMainFrame::OnVfwPreviewvideo()
{
 //TODO: 在此添加命令处理程序代码
 CAPTUREPARMS CapParms;
 bPreview =! bPreview;                        //捕获预览开关
```

```
    if(bPreview){
        capCaptureGetSetup(m_hWndCap,&CapParms,sizeof(CAPTUREPARMS)); //获取参数
        CapParms.dwIndexSize=324000;
        CapParms.fMakeUserHitOKToCapture=!CapParms.fMCIControl;
        CapParms.wPercentDropForError=100;        //丢帧百分比
        CapParms.wNumVideoRequested=5;            //视频缓冲区最大数量
        CapParms.wChunkGranularity=0;
        CapParms.fYield=TRUE;                     //产生后台线程进行视频捕捉
        CapParms.fCaptureAudio=FALSE;             //不捕获音频
        CapParms.vKeyAbort=0;                     //终止捕获的键
        CapParms.fAbortLeftMouse=CapParms.fAbortRightMouse=FALSE;
                                                  //鼠标左右键控制停止捕捉
        CapParms.dwRequestMicroSecPerFrame=1000000/25;
                                                  //捕获的帧率25f/s
        capSetCallbackOnYield(m_hWndCap,NULL);
                                                  //回调函数为空
        capCaptureSetSetup(m_hWndCap,&CapParms,sizeof(CAPTUREPARMS));
                                                  //配置捕捉参数
        //如果要捕获视频流，则要使用函数指定不生成文件。否则将会自动生成AVI文件
        capCaptureSequenceNoFile(m_hWndCap);
        m_vfwState = PREVIEW;
    }else{
        capCaptureAbort(m_hWndCap);
    }
}
```

上述程序实现捕获预览视频，CAPTUREPARMS 结构体包含了捕获参数，包括捕获的帧率、视频缓冲区最大数量、鼠标左右键控制停止捕获等参数。capCaptureSequenceNoFile 启动捕获、预览。

关闭 VFW 视频采集，重载窗口关闭消息 WM_CLOSE，添加处理函数。实现代码如下。

```
void CMainFrame::OnClose()
{
 //TODO：在此添加消息处理程序代码和/或调用默认值
 capCaptureAbort(m_hWndCap);                          //终止捕获
 capDriverDisconnect(m_hWndCap);                      //断开与驱动的连接
 Sleep(100);                                          //等待前面的操作完毕
 capSetCallbackOnError(m_hWndCap,NULL);               //注销回调函数
 capSetCallbackOnStatus(m_hWndCap,NULL);              //注销回调函数
 capSetCallbackOnVideoStream(m_hWndCap,NULL);         //注销回调函数
 delete lpbiIn;
 Sleep(100);                                          //确保前面的操作完毕
 CFrameWnd::OnClose();
}
```

关闭 VFW 视频采集首先终止捕获，然后断开与驱动的连接，确保这些操作完成后，注销各种已注册的回调函数，删除申请的内存，关闭应用程序。

至此我们就可以在初始化 VFW 视频设备后，预览捕获到的视频数据，如图 10-4 所示。

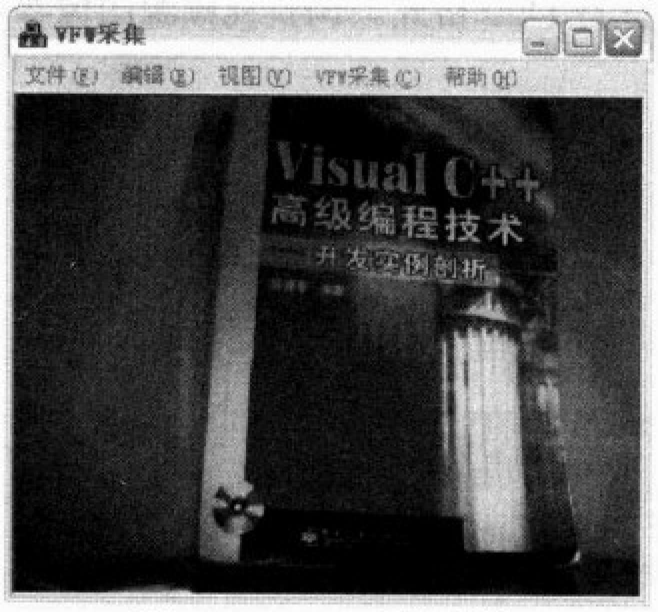


图 10-4　VFW 采集预览

10.2.6　配置视频格式及图像参数

视频采集设备提供了默认的视频格式和图像参数，也可以自行配置。VFW 提供配置视频采集设备的视频格式和图像参数。即函数：

```
capDriverGetCaps(m_hWndCap,&m_caps,sizeof(CAPDRIVERCAPS));
```

m_hWndCap 是 VFW 窗口句柄，m_caps 是指向 CAPDRIVERCAPS 结构体，最后一个参数是该结构体的大小。

根据 m_caps 的内容判断采集设备驱动器的性能，配置视频格式和图像源参数。对项目 vfwapp 的“VFW 采集”菜单子项“配置视频格式”，添加消息处理函数。实现代码如下。

```
void CMainFrame::OnVfwVideoformat()
{
 //TODO: 在此添加命令处理程序代码
 DWORD fsize;
 if(m_caps.fHasDlgVideoFormat){
        capDlgVideoFormat(m_hWndCap);                          //显示视频格式对话框
        fsize = capGetVideoFormatSize(m_hWndCap);              //读取新的格式大小
        capGetVideoFormat(m_hWndCap, lpbiIn, fsize);           //读取新格式数据
 }
 if(m_caps.fHasDlgVideoSource){
        capDlgVideoSource(m_hWndCap);                //显示图像源参数：亮度、色度和饱和度等
 }
}
```

首先判断该视频驱动是否有视频格式对话框，若有则显示视频格式对话框，读取新的格式大小，读取新格式数据。若视频驱动支持视频图像源参数配置对话框，则显示该对话框。

视频格式配置视频图像的分辨率、像素深度和压缩参数，如图 10-5 所示。

分辨率列出了当前视频采集设备所支持的所有公式，像素深度列出了图像数据格式，如

352 像素×288 像素、320 像素×240 像素、176 像素×166 像素等。普通的摄像头一般只支持 RGB24，较高级的采集设备支持 I420 格式。调整视频源图像参数，如图 10-6 所示。

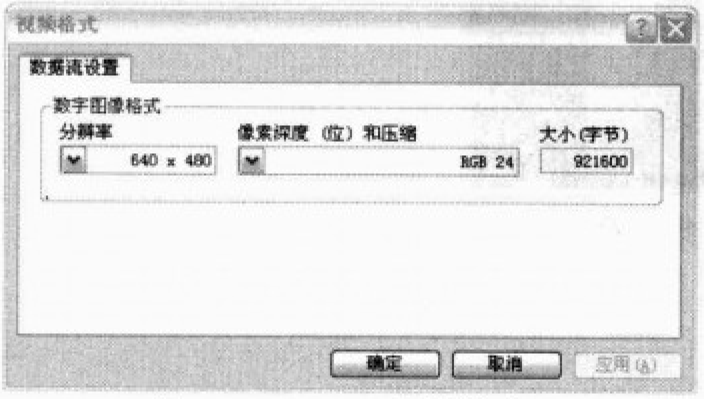


图 10-5　VFW 视频格式

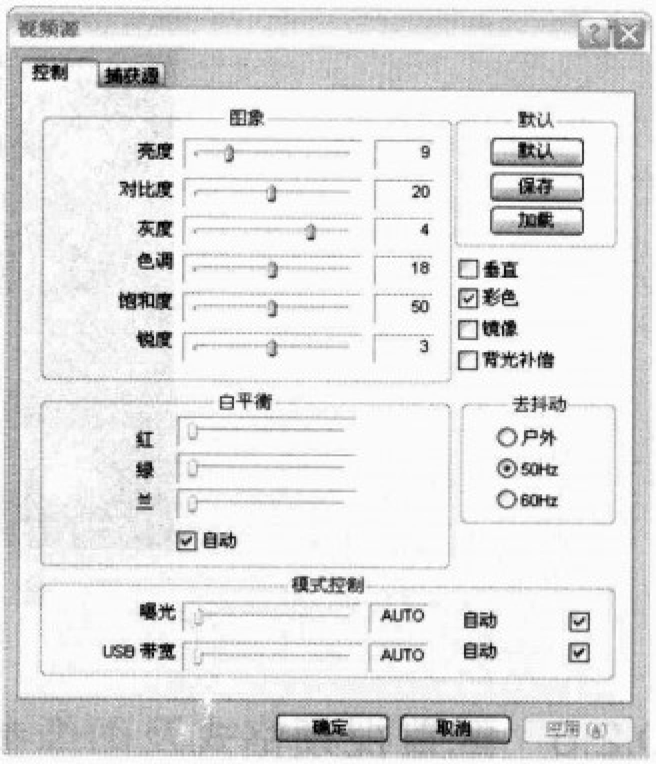


图 10-6　视频图像参数

图像参数包括图像、白平衡、模式控制等。对这些参数的调整程序可以马上作用于采集的视频画面，即时调整即时作用于画面。

10.3　使用系统插件实现视频编/解码

基于上述平台的第一个应用是使用系统中已注册的编/解码插件（VFW 格式），对捕获的视频数据进行编码、解码处理，并对编码后的数据写 AVI 文件存盘。使用系统编/解码插件，仍然是 VFW 编程开发，即使用其 ICM（Installable Compression Manager）压缩器管理模块，调用相应的视频 CODEC 算法完成视频编/解码。

ICM 是介于应用程序与编/解码算法库之间，帮助编程开发者管理用于视频编码压缩和解码的 CODEC 编/解码器。ICM 提供了应用程序对软件编码、解码的调用接口，通过它提供的各种函数、消息和宏，应用程序可以实现视频的实时编码压缩、解码等功能，其中主要包括编/解码库安装、选择和打开、压缩器的配置、编码和解码等功能。

其实，具体实现编/解码的是 CODEC 编/解码算法库。CODEC 编/解码算法，如 MPEG4 的 CODEC、H.263 的 CODEC。自己编写算法来实现视频编/解码是一个非常复杂、烦琐的工作，我们可以使用操作系统自带的 CODEC，并且很多 CODEC 软件提供商提供了直接安装的 Install 程序。如 XviD 的 xvid.dll，在 xvid.inf 上单击鼠标右键，选择安装操作。安装完毕后，通过 AVICap 提供的高层接口宏 capDlgVideoCompression 显示的对话框的下拉列表中，就会出现“XviD MPEG-4 Video Codec”的选项供用户选择调用，并提供设置选项可以对压缩参数进行配置。光盘提供了 xvid.dll 和 xvid.inf 两个文件，读者也可以到网上下载。

10.3.1 使用系统视频编/解码插件

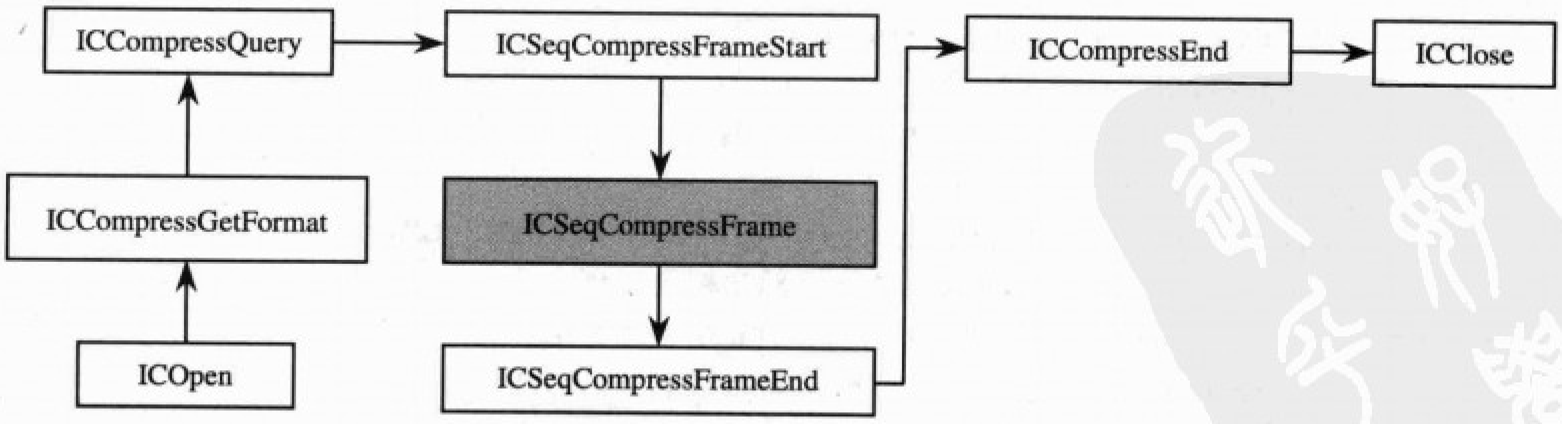
我们在使用编/解码插件实现视频编码、视频解码并送窗口显示的基础上，增加存盘 AVI 文件功能。在使用 ICM 管理编/解码插件的过程中，编码和解码使用过程几乎相同。工作过程："打开"→"准备"→"循环使用"→"结束"→"关闭"，ICM 函数如表 10-3 所示。

表 10-3 ICM 函数列表

函数名	功能说明
ICOpen	打开编码器或解码器
ICClose	关闭编码器或解码器
ICDecompressBegin	准备解码
ICDecompress	反复调用，解码图像
ICDecompressEnd	解码结束
ICSeqCompressFrameStart	准备序列编码
ICSeqCompressFrame	反复调用，编码序列
ICSeqCompressFrameEnd	序列编码结束
ICCompressEnd	编码终止
ICCompressGetFormat	获取编码输出格式
ICCompressQuery	查询从源到目的格式是否支持

编/解码函数调用过程如图 10-7 所示，其中有底纹的模块为循环调用。

ICM 解码函数调用过程（a）



ICM 编码函数调用过程（b）

图 10-7 ICM 编/解码函数调用过程

由上图分析可以看出，ICM 的编/解码插件在使用时基本是按照打开插件、循环使用插件和最后关闭插件的过程进行的，只是编码函数的使用更为复杂。

编程实现ICM函数调用。对项目vfwapp的“VFW采集”菜单子项“捕获编/解码”，添加消息处理函数，具体的代码如下。

```
/* 对捕获的视频做编码、解码处理*/
void CMainFrame::OnVfwCodec()
{
  // TODO: 在此添加命令处理程序代码
  DWORD fsize;
  /* 打开xvid编码和解码 */
  hic1  =  ICOpen(mmioFOURCC('v','i','d','c'),    mmioFOURCC('X','V','I','D'),
ICMODE_COMPRESS);
  if (hic1 == 0)
         AfxMessageBox(_T("打开编码器失败!"));
  hic2  =  ICOpen(mmioFOURCC('v','i','d','c'),    mmioFOURCC('X','V','I','D'),
ICMODE_DECOMPRESS);
  if (hic2 == 0)
         AfxMessageBox(_T("打开解码器失败!"));
  //获取VFW视频捕获格式大小
  fsize = capGetVideoFormatSize(m_hWndCap);
  capGetVideoFormat(m_hWndCap, lpbiIn, fsize);
  //初始化AVI写文件
  InitAVIWriteOpt();
  //添加BMP头结构
  lpbiOut->bmiHeader.biSize          = sizeof(BITMAPINFOHEADER);
  lpbiOut->bmiHeader.biWidth         = lpbiIn->bmiHeader.biWidth;
  lpbiOut->bmiHeader.biHeight        = lpbiIn->bmiHeader.biHeight;
  lpbiOut->bmiHeader.biPlanes        = 1;
  lpbiOut->bmiHeader.biBitCount      = 24;
  lpbiOut->bmiHeader.biCompression   = BI_RGB;
  lpbiOut->bmiHeader.biSizeImage     = lpbiIn->bmiHeader.biWidth*lpbiIn->
bmiHeader. biHeight*3;
  lpbiOut->bmiHeader.biXPelsPerMeter = 0;
  lpbiOut->bmiHeader.biYPelsPerMeter = 0;
  lpbiOut->bmiHeader.biClrUsed       = 0;
  lpbiOut->bmiHeader.biClrImportant  = 0;
  //根据输入视频的格式，获取输出格式
  if (ICCompressGetFormat(hic1,lpbiIn,lpbiTmp)!=ICERR_OK)
       AfxMessageBox(_T("编码器不能读取输出格式! "));
  if (ICCompressQuery(hic1,lpbiIn,lpbiTmp) != ICERR_OK)
       AfxMessageBox(_T("不能处理编码器输入输出格式! "));
  //设置编码压缩参数
  pc.cbSize          = sizeof(COMPVARS);                 //结构体大小
  pc.dwFlags         = ICMF_COMPVARS_VALID;
  pc.hic             = hic1;                             //编码器句柄
  pc.fccType         = mmioFOURCC('v','i','d','c');
  pc.fccHandler      = mmioFOURCC('X','V','I','D');
```

```
    pc.lpbiOut          = lpbiTmp;                              //输出格式
    pc.lKey             = 100;                                  //key帧频率
    pc.lQ               = 10000;                                //图像质量
   //开启视频序列编码
    if(!ICSeqCompressFrameStart(&pc, lpbiIn))
          return;
   //开启视频解码
    ICDecompressBegin(hic2,lpbiTmp,lpbiOut);
    m_vfwState  = ENCDEC;
   }
```

上述程序实现获取 xvid 视频编/解码 CODEC，获取 VFW 视频捕获格式大小、初始化 AVI 写文件操作、设置视频编码参数信息，最后开启视频的编码和解码。

10.3.2 视频编码码流写AVI文件

AVI 文件格式以通用和跨平台等特点被许多影音文件所采用。AVI 是一个多媒体容器，它可以只有视频，也可以音/视频都有。

在写 AVI 文件中，定义局部变量：

```
    PAVIFILE          m_pFile;              //AVI文件
    AVISTREAMINFO     strhdr;               //AVI流信息
    PAVISTREAM        ps;                   //AVI流指针
   DWORD              m_Frame;              //记录帧数
```

使用标准打开文件对话框，输入要保存的 AVI 文件名，初始化基本操作。开启 VFW 捕捉视频，具体代码如下。

```
   void CMainFrame::InitAVIWriteOpt()
   {
    CString filename;
    CFileDialog FileDlg(FALSE,_T("avi"));                   //另存为对话框，文件类型为AVI
    if (FileDlg.DoModal()==IDOK)
    {
          filename = FileDlg.GetPathName();                 //AVI文件名及路径
          m_Frame = 0 ;                                     //视频编码序号
          AVIFileInit() ;                                   //AVI文件初始化
          bSaveAVI = TRUE;
          //打开AVI文件
          AVIFileOpen(&m_pFile,filename,OF_WRITE | OF_CREATE,NULL);
          memset(&strhdr, 0, sizeof(strhdr)) ;              //结构体清空
          strhdr.fccType    = streamtypeVIDEO;              //流类型为视频
          strhdr.fccHandler  = 0;                           //视频流不压缩
          strhdr.dwScale    = 1;                            //数据量大小
          strhdr.dwRate     = 25;                           //帧率为25f/s
          strhdr.dwSuggestedBufferSize = lpbiIn->bmiHeader.biSizeImage;
                                                            //建议的缓冲区大小
```

```
            //设置矩形区域
            SetRect(&strhdr.rcFrame,  0,  0,  lpbiIn->bmiHeader.biWidth,  lpbiIn->
bmiHeader.biHeight);
            ps = NULL;
            //创建AVI文件流
            AVIFileCreateStream(m_pFile,&ps,&strhdr);
            //开始捕捉
            capCaptureSequenceNoFile(m_hWndCap);
     }
    }
```

上述程序实现对写 AVI 文件做必要的初始化工作。AVIFileInit 是标准的 VC 函数，用来初始化并打开 AVI 文件，然后对 AVISTREAMINFO 结构体初始化。设置视频区域，创建 AVI 文件流，最后开始捕捉视频。

在回调函数 VideoCallbackProc 中，对编码后的码流写 AVI 文件。实现代码如下。

```
    LRESULT FAR PASCAL VideoCallbackProc(HWND hWnd, LPVIDEOHDR lpVHdr)
    {
     unsigned char *bufi, *buf;
     bufi = new unsigned char[lpVHdr->dwBytesUsed+40];    //原图像
     buf = new unsigned char[lpVHdr->dwBytesUsed];          //编码压缩码流
     memcpy((void *)(bufi), lpVHdr->lpData, lpVHdr->dwBytesUsed);
     unsigned char *buf1;
     buf1 = buf;
     if (m_vfwState==ENCDEC) { //编码和解码
        //Encode
           //使用ICM编码器进行图像编码
           buf1  =  (unsigned  char*)ICSeqCompressFrame(&pc,0,bufi,  &IsKeyFrame,&
FrameSize);
           if (bSaveAVI){
                  AVIStreamSetFormat(pMainFrame->ps,            //设置AVI文件流格式
                                    pMainFrame->m_Frame++,
                                    lpbiTmp,sizeof(BITMAPINFO));
                  // 写AVI数据流到文件
                  AVIStreamWrite(pMainFrame->ps,                //打开的avi流
                               pMainFrame->m_Frame,             //要写的第一帧
                               1,                               //要写的帧数
                               (LPBYTE)buf1,                    //要写的数据
                          lpbiTmp->bmiHeader.biSizeImage,       //要写的数据的大小
                               AVIIF_KEYFRAME,NULL,NULL);       //关键帧
           }
     //Decode
           //使用ICM解码器进行解码
           ICDecompress(hic2,0,&lpbiTmp->bmiHeader,buf1,&lpbiOut->bmiHeader,&bufo
[40]);
     } else { //不做编/解码，直接把图像送显示
          memcpy(&bufo[40],bufi,lpVHdr->dwBytesUsed);
```

```
    }
        //激活显示窗口，发消息显示图像。
    pMainFrame->GetActiveView()->InvalidateRect(NULL,FALSE);
    delete bufi;
    delete buf;
    return (LRESULT) TRUE;
}
```

上述回调函数使用 ICM 的编码器进行视频编码，然后把码流写到 AVI 文件中，然后使用 ICM 解码器进行解码，最后把解码的图像送显示。

10.3.3 关闭平台

除关闭 VFW 驱动器、断开与采集设备的连接外，释放申请的内存，还需关闭 AVI 文件相关操作，实现代码如下。

```
void CMainFrame::OnClose()
{
 //TODO: 在此添加消息处理程序代码和/或调用默认值
 capCaptureAbort(m_hWndCap);                          //终止捕获
 capDriverDisconnect(m_hWndCap);                      //断开与驱动的连接
 Sleep(100);                                          //确保前面的操作完成
 capSetCallbackOnError(m_hWndCap,NULL);               //注销出错回调函数
 capSetCallbackOnStatus(m_hWndCap,NULL);              //注销状态回调函数
 capSetCallbackOnVideoStream(m_hWndCap,NULL);         //注销视频流处理回调函数
 delete lpbiIn;
 delete lpbiTmp;
 delete lpbiOut;
 if (m_vfwState==ENCDEC){                             //捕获编/解码
       ICDecompressEnd(hic2);                         //终止解码操作
       ICClose(hic2);                                 //关闭解码器
       ICSeqCompressFrameEnd(&pc);                    //终止序列编码
       ICCompressEnd(hic1);                           //终止编码操作
       ICClose(hic1);                                 //关闭编码器
       AVIStreamClose(ps);                            //关闭AVI流
       if(m_pFile != NULL)
           AVIFileRelease(m_pFile);                   //关闭AVI文件
 }
 Sleep(100);
 CFrameWnd::OnClose();
}
```

退出程序前，首先终止捕获视频，然后断开与 VFW 驱动的连接。确保这些操作成功后，分别进行注销回调函数、关闭 ICM 编/解码器、关闭 AVI 文件流、关闭 AVI 文件等操作。

10.4 使用XviD CODEC算法实现视频编/解码

平台应用一使用了系统插件实现视频编/解码，该应用对我们来说视频流是“透明”的。我们是通过标准的开发框架函数如 ICM 控制视频设备、捕捉视频流、枚举编/解码器和使用 ICM 控制 CODEC 编/解码器。

另外，在使用 VFW 进行视频开发时，需要对捕获的视频流直接处理，如编码、滤波和识别等。现在我们在 VFW 的视频捕获、预览的平台下，使用 Xvid CODEC 算法对视频流做编码和解码。

10.4.1 生成XviD算法静态库（编译过程）

XviD 是开源的 MPEG-4 视频编/解码算法，网络上的许多最新电影和影音文件都使用了 XviD 算法压缩。另外，做 MPEG-4 视频硬件编码的工程师也都几乎选择了 XviD 作为蓝本，算法支持流行的 I420（YUV420）视频格式编码。由于 XviD 的最初编译平台是 VC6.0，所以在使用 VC++ 2005 编译 XviD 时会出现许多错误。

- 首先把 xvidcore 1.1.3 的 VC 6.0 工程转向 VC 2005 项目。打开*.dsw 工作区文件，默认确定 VC 2005 的设置。
- 特别注意：xvidcore 1.1.3 项目在硬盘中的路径名中（文件夹），不得有空格，否则在编译*.asm 汇编文件时不成功。
- 确保汇编编译器 nasm.exe 添加到 VC 2005 的编译环境或路径中。方法一：在 VC 2005 开发环境下依次单击“工具”→“选项”→“项目和解决方案”→“VC++目录可执行文件”按钮，添加 nasm 所在的文件夹路径。方法二：直接把 nasm.exe 程序复制到 VC 2005 的 bin 目录下，另外注意汇编编译器的名字必须是 nasm.exe。如果读者从网上下载了 nasmw.exe，则需要手工修改。
- 选中 libxvidcore_static 项目，编译链接。

出现错误	1>Assembling xxx\src\image\x86_asm\colorspace_yuyv_mmx.asm 1>Assembling xxx\src\image\x86_asm\colorspace_yuyv_mmx.asm 1>nasm: error: no input file specified 1>type `nasm -h' for help
错误原因	因为在 VC 6.0 工程转换到 VC 2005 项目的时候，开发环境在命令行里的宏给自动加上了双引号，如 nasm -f win32 -DPREFIX -I"$(InputDir)"\ -o $(IntDir)\$(InputName).obj $(InputPath)
问题解决	只需要在出错文件的“属性”→“用户自定义生成步骤”→“常规”→“命令行”里面把-I"$(InputDir)"改成 -I$(InputDir)就行了
出现错误	error C2632：“int”后面的“int”非法
错误原因	xvid 为了方便平台间的移植，在 src/portab.h 中定义了一个 intptr_t 宏。但是 VC 2005 已经定义了
问题解决	在 src/portab.h 找到 intptr_t 的定义，并注释掉该行

- 在反复修改项目、编译过程后，当重新生成方案时，会出现“vc80.idb 无法删除”的现象。解决方法：把 debug 目录下的该文件手工删除，或者重启 VC 2005，问题还不能解决就需要重启计算机了，或许是 VC 2005 编译器的问题，笔者就曾遇到过此类问题。
- 至此 libxvidcore_static 项目应该已经编译通过，生成 libxvidcore.lib。另外，使用 VC 6.0 编译生成的库不能被 VC 2005 调用，使用 VC 2005 编译生成的库才能被 VC 2005 项目使用。

10.4.2 实现XviD CODEC视频编码算法函数

XviD 项目中提供的例程展示了如何使用 XviD CODEC 算法。视频编码应用是 xvid_encraw，解码应用是 xvid_decraw。我们根据这两个例程应用添加 XviD CODEC 算法库到 vfwapp 项目中，实现 VFW 的实时采集视频，然后 xvid 编码、xvid 解码，最后显示解码后的图像。

把使用 XviD CODEC 算法的程序放在文件 xvid_codec.cpp、xvid_codec.h 中。注意算法使用函数使用的是 C 语言，但实现文件命名一定要以.cpp 结尾，C 结尾则会出现编译错误。

引入 XviD CODEC 算法库 libxvidcore.lib。VC 2005 开发环境若提示“找不到 libcd.lib”的编译错误，则通过“配置属性”→“链接器”→“输入”→“忽略特定库”命令添加“libcd.lib”。

定义头文件 xvid_codec.h，定义编码参数及使用 XviD CODEC 算法的函数，实现代码如下。

```
#ifndef _ENC_DEC_HEADER_
#define _ENC_DEC_HEADER_
#ifdef __cplusplus
extern "C" {
#endif
/*编码器用到的编码参数*/
typedef struct  {
 int width;                //编码图像宽度
 int height;               //编码图像高度
 int max_key_interval;     //关键帧间隔
 int framerate;            //编码帧率
 int quant;                //量化步长
 int bitrate;              //码流大小，实现CBR
 void *handle;             //编码器句柄
} ENC_PARAM;
int enc_init(ENC_PARAM *enc_param);          //根据传入的编码参数，创建编码器
int enc_stop();                              //销毁编码器
/*编码一帧图像*/
int enc_main(unsigned char *image,           //编码图像指针
             unsigned char *bitstream,       //编码后码流指针
             int *key,                       //关键帧标志
             int *stats_type,                //编码类型
```

```
        int *stats_quant,                    //量化步长
        int *stats_length);                  //编码后码流大小
int dec_init(int width, int height);         //创建解码器
int dec_stop();                              //销毁解码器
/*解码一帧视频帧*/
int dec_main(unsigned char *istream,         //码流指针
        unsigned char *ostream,              //解码图像的指针
        int istream_size,                    //码流长度
        int width);                          //图像宽度
/*图像颠倒*/
void SwapBitmap(unsigned char *pInBuf, unsigned char *pOutBuf, int nPicType);
#ifdef __cplusplus
}
#endif
#endif
```

上述程序是 XviD 视频编/解码算法的接口。编码时首先根据参数创建编码器，然后循环使用 enc_main 对图像进行编码，最后退出编码器并销毁编码器。解码时同样创建解码器，循环使用 dec_main 对图像进行解码，最后退出解码器并销毁解码器。由于 BMP 位图的数据是颠倒存放的，所以显示图像时需要对采集的图像进行颠倒操作。

另外，XviD CODEC 算法的一些其他编码参数在 enc_init、enc_main 中做了默认设置。读者可以根据需求作改动，如使用变长码流设置 quant；图像编码质量 ARG_QUALITY 使用大于 2 等。

初始化 XviD CODEC 编码器，函数 enc_init，根据设定的编码参数初始化编码器，返回编码器句柄，后面的所有控制都是通过该句柄来完成的。实现代码如下。

```
/* 第一次使用，需要初始化编码器，把预设参数传给编码器*/
int enc_init(ENC_PARAM *pParam)
{
 int xerr;
  xvid_plugin_single_t single;        //单个插件，即码流控制
 xvid_enc_plugin_t plugins[7];        //所有的插件
 xvid_gbl_init_t xvid_gbl_init;       //全局初始化
 xvid_enc_create_t xvid_enc_create;   //编码结构体
 /*XVID CORE初始化*/
 /*设置版本，XVID CORE会检查版本*/
 memset(&xvid_gbl_init, 0, sizeof(xvid_gbl_init));
 xvid_gbl_init.version = XVID_VERSION;
     xvid_gbl_init.debug = 0;         //ARG_DEBUG;
 if (1) {                             /* 使用汇编优化*/
#ifdef ARCH_IS_IA64
       xvid_gbl_init.cpu_flags = XVID_CPU_FORCE | XVID_CPU_ASM;
#else
       xvid_gbl_init.cpu_flags = 0;
#endif
 } else {
```

```
        xvid_gbl_init.cpu_flags = XVID_CPU_FORCE;
    }
    /* 初始化XVID CODEC ——每次编码只能调用一次*/
    xvid_global(NULL, XVID_GBL_INIT, &xvid_gbl_init, NULL);
    /*XVID 版本*/
    memset(&xvid_enc_create, 0, sizeof(xvid_enc_create));
    xvid_enc_create.version = XVID_VERSION;
    /* 输入视频帧的宽和高*/
    xvid_enc_create.width = XDIM = pParam->width;
    xvid_enc_create.height = YDIM = pParam->height;
    xvid_enc_create.profile = XVID_PROFILE_AS_L4;
    xvid_enc_create.plugins = plugins;
    xvid_enc_create.num_plugins = 0;
    if (ARG_SINGLE)                              //使用码流控制
    {
        memset(&single, 0, sizeof(xvid_plugin_single_t));
        single.version = XVID_VERSION;
        single.bitrate = pParam->bitrate;  //设置码流大小
        plugins[xvid_enc_create.num_plugins].func = xvid_plugin_single;
        plugins[xvid_enc_create.num_plugins].param = &single;
        xvid_enc_create.num_plugins++;
    }
    xvid_enc_create.num_threads = 0;
    if (1) {                /*视频帧率*/
        xvid_enc_create.fincr = 1;
        xvid_enc_create.fbase = pParam->framerate;
    } else {
        xvid_enc_create.fincr = FRAMERATE_INCR;
        xvid_enc_create.fbase = (int) (FRAMERATE_INCR * ARG_FRAMERATE);
    }
  if (1) {/*最大关键帧间隔*/
        xvid_enc_create.max_key_interval = pParam->max_key_interval;
        }else {
       xvid_enc_create.max_key_interval = (int) ARG_FRAMERATE *10;
       }
     xvid_enc_create.global = 0;           /* 编码器全局选项*/
  xerr = xvid_encore(NULL, XVID_ENC_CREATE, &xvid_enc_create, NULL);
   enc_handle = xvid_enc_create.handle;     /* 从结构体中取得编码器实例句柄*/
   return (xerr);
  }
```

上述程序是XviD编码器的初始化，首先全局初始化，包括函数、内存等的初始化。根据编码器的参数创建编码器，返回编码器的实例句柄，后续的编码控制都基于该句柄。

XviD CODEC编码视频函数。对输入的视频帧进行编码，返回编码后的码流长度，同时返回帧类型和量化步长。实现代码如下。

```
  int enc_main(unsigned char *image, unsigned char *bitstream, int *key, int
```

```
*stats_type,
            int *stats_quant, int *stats_length)
    {
     int ret;
     xvid_enc_frame_t xvid_enc_frame;                      //图像帧编码参数
     xvid_enc_stats_t xvid_enc_stats;                      //编码器的状态
     memset(&xvid_enc_frame, 0, sizeof(xvid_enc_frame)); //清空图像编码结构体
     xvid_enc_frame.version = XVID_VERSION;                //XVID编码器的版本
     memset(&xvid_enc_stats, 0, sizeof(xvid_enc_stats)); //编码器的状态结构体清空
     xvid_enc_stats.version = XVID_VERSION;                //编码器状态结构体的版本
    xvid_enc_frame.bitstream = bitstream;                  //初始化码流缓冲区指针
     xvid_enc_frame.length = -1;
     if (image) {/* 输入图像、格式初始化*/
           xvid_enc_frame.input.plane[0] = image;          //待编码图像的指针
           xvid_enc_frame.input.csp = XVID_CSP_I420;       //图像颜色格式
           xvid_enc_frame.input.stride[0] = XDIM;          //图像宽度
     } else {
           xvid_enc_frame.input.csp = XVID_CSP_NULL;       //没有编码图像
     }
     /* 设置编码器的VOL参数*/
     xvid_enc_frame.vol_flags = 0;
     /* 设置编码器的VOP参数*/
     xvid_enc_frame.vop_flags = picture_pixels[ARG_QUALITY];
     /* 编码帧类型——让编码器自动判决*/
     xvid_enc_frame.type = XVID_TYPE_AUTO;
     /* 强制量化步长，在CORE内使用码流控制管理quant*/
     xvid_enc_frame.quant = 0;
     /* 设置运动估计参数*/
     xvid_enc_frame.motion = ME_tables[ARG_QUALITY];
     /* 编码视频帧*/
     ret = xvid_encore(enc_handle, XVID_ENC_ENCODE, &xvid_enc_frame,
                         &xvid_enc_stats);
     *key = (xvid_enc_frame.out_flags & XVID_KEYFRAME);
     *stats_type = xvid_enc_stats.type;        //帧编码类型I/P/B
     *stats_quant = xvid_enc_stats.quant;      //编码量化步长，由RC控制产生
     *stats_length = xvid_enc_stats.length;   //编码后的码流长度，单位字节B
     return (ret);
    }
```

上述程序实现实际的图像编码功能。编码参数有编码帧类型、量化步长、运动估计参数等。编码后返回当前编码帧类型、量化步长和图像编码后的码流长度。

XviD CODEC 算法编码器销毁，释放所有内存。实现代码如下。

```
    int enc_stop()
    {
     int xerr;
     /* 销毁编码器实例*/
```

```
  xerr = xvid_encore(enc_handle, XVID_ENC_DESTROY, NULL, NULL);
  return (xerr);
}
```

上述程序实现销毁编码器，释放申请的内存。

10.4.3 实现XviD CODEC视频解码算法函数

为观察编码后的图像效果或解码 MPEG-4 视频文件，实现使用 XviD CODEC 解码器的函数。另外修改了源 XviD CORE 工程的解码属性，直到解码完一帧图像才返回。解码 VOL 头后不再返回。

创建解码器，dec_init 函数实现。创建解码器句柄，解码器的所有控制都是通过该句柄来实现的。初始化一大特点就是需预先知道图像的宽和高，源 XviD CORE 算法中可以从码流中分析出图像的宽和高。具体代码如下。

```
/* 在第一次使用前，调用该函数初始化解码器*/
int dec_init(int width, int height)
{
 int ret;
 xvid_gbl_init_t  xvid_gbl_init;                          //全局初始化
 xvid_dec_create_t xvid_dec_create;                        //解码结构体
 memset(&xvid_gbl_init, 0, sizeof(xvid_gbl_init_t));       //清空全局初始化结构体
 memset(&xvid_dec_create, 0, sizeof(xvid_dec_create_t));   //清空解码结构体
 /*------------------------------------------------------------------------------
  * xvid core 初始化
  *----------------------------------------------------------------------------*/
 xvid_gbl_init.version = XVID_VERSION;
 if(1)/*强制使用汇编优化*/
#ifdef ARCH_IS_IA64
      xvid_gbl_init.cpu_flags = xvid_CPU_FORCE | xvid_CPU_IA64;
#else
 xvid_gbl_init.cpu_flags = 0;
#endif
 else
      xvid_gbl_init.cpu_flags = XVID_CPU_FORCE;
 xvid_gbl_init.debug = 0;          //无调试信息
 xvid_global(NULL, 0, &xvid_gbl_init, NULL);//全局初始化
 /*------------------------------------------------------------------------------
  * xvid decoder 初始化
  *----------------------------------------------------------------------------*/
 /* Version */
 xvid_dec_create.version = XVID_VERSION;
 /*图像宽和高，由解码器输入*/
 xvid_dec_create.width = width;              //图像的宽
 xvid_dec_create.height = height;            //图像的高
```

```
    ret = xvid_decore(NULL, XVID_DEC_CREATE, &xvid_dec_create, NULL);
    dec_handle = xvid_dec_create.handle;     //返回解码器实例句柄
    return(ret);
}
```

上述程序实现创建解码器，初始化函数、申请内存空间等。返回解码器实例句柄。

使用 XviD CODEC 视频解码器，对输入的码流解码，返回已经解码的码流长度。解码器可以选择是否对图像做去除“块效应”。目前 XviD CODEC 算法仅支持解码输出 I420 格式，不支持 RGB 格式。

```
/*解码一帧图像*/
int dec_main(unsigned char *istream, unsigned char *ostream, int istream_size,
int width)
{
    int ret;
    xvid_dec_frame_t xvid_dec_frame;
    /* 清零，变量状态确定*/
    memset(&xvid_dec_frame, 0, sizeof(xvid_dec_frame_t));
    memset(xvid_dec_stats, 0, sizeof(xvid_dec_stats_t));
    /* 设置版本*/
    xvid_dec_frame.version = XVID_VERSION;
    xvid_dec_stats->version = XVID_VERSION;
    /* 解码标志清零*/
    xvid_dec_frame.general = 0;
    /*解码后的图像是否做去块效应，可以对Y和UV分开处理*/
    xvid_dec_frame.general |= xvid_DEBLOCKY;// | xvid_DEBLOCKUV;
    /* 输入码流*/
    xvid_dec_frame.bitstream  = istream;
    xvid_dec_frame.length = istream_size;
    /* 输出图像结构体*/
    xvid_dec_frame.output.plane[0]  = ostream;
    xvid_dec_frame.output.stride[0] = width;
    xvid_dec_frame.output.csp = XVID_CSP_I420;//CSP;
    ret    =    xvid_decore(dec_handle,    XVID_DEC_DECODE,    &xvid_dec_frame,
xvid_dec_stats);
    return(ret);
}
```

上述程序实现解码一帧图像，传入码流和长度，返回解码后的图像。函数返回值为当前已经解码的码流长度大小。

XviD CODEC 算法解码器销毁，释放内存。

```
/* 关闭解码器，释放资源*/
int dec_stop()
{
    int ret; //返回值
    //销毁解码器，释放申请的内存等
```

```
 ret = xvid_decore(dec_handle, XVID_DEC_DESTROY, NULL, NULL);
 return(ret);
}
```

销毁解码器，销毁解码器申请的内存空间。

在 XviD CODEC 算法编码器和解码器中，实例句柄为全局变量。

```
#define IMAGE_SIZE(x,y) ((x)*(y)*3)
static void *enc_handle = NULL;  //编码器实例句柄
static void *dec_handle = NULL;  //解码器实例句柄
```

编码器和解码器的所有控制均通过实例句柄完成，因此如果是多路的编码和解码，则有多个编码器句柄或解码器句柄。

10.4.4 使用XviD CODEC编/解码算法

现在开始使用 XviD CODEC 视频编/解码算法，实现 VFW 实时采集视频、XviD 编/解码，显示解码后的图像。

项目 vfwapp_2 中添加自定义函数，进行配置 XviD CODEC 编/解码器。具体代码如下。

```
/*配置XVID CODEC视频编/解码算法*/
void CMainFrame::Config_XVIECODEC()
{
 ENC_PARAM m_Param;
 m_Param.width = lpbiIn->bmiHeader.biWidth;   //视频图像宽度
 m_Param.height= lpbiIn->bmiHeader.biHeight;  //视频图像高度
 m_Param.bitrate = 300000;                    //码流大小，使用码流控制
 m_Param.max_key_interval = 100;              //关键帧间隔
 m_Param.framerate = 25;                      //帧率
 m_Param.quant = 0;                           //量化步长，0表示使用码流控制
 enc_init(&m_Param);                          //创建XviD CODEC编码器
 dec_init(m_Param.width,m_Param.height);      //创建XviD CODEC解码器
}
```

上述程序实现配置编码器的编码参数，包括图像的宽和高、码流大小、关键帧间隔、帧率。创建编码器实例，根据图像的宽和高创建解码器实例。

在视频格式配置中，调用函数 Config_XVIECODEC 完成编码器的创建工作，注意配置视频格式的颜色空间为 I420，编码器不支持 RGB 格式。实现代码如下。

```
void CMainFrame::OnVfwVideoformat()
{
 // TODO: 在此添加命令处理程序代码
 DWORD fsize;
 if(m_caps.fHasDlgVideoFormat){
      capDlgVideoFormat(m_hWndCap);
      fsize = capGetVideoFormatSize(m_hWndCap);
      capGetVideoFormat(m_hWndCap, lpbiIn, fsize);
```

```
        Config_XVIECODEC();
    }
}
```

上述程序首先判断 VFW 驱动是否支持视频格式对话框，若支持则显示对话框，然后获取格式大小，获取视频图像格式。根据用户选择的参数配置视频编/解码器。

修改 VFW 视频回调函数 VideoCallbackProc，调用 XviD CODEC 编/解码器的编码函数 enc_main 和解码函数 dec_main，完成对捕获到的视频帧的编码、解码。另外，由于该编/解码器仅支持 I420，所以解码后的图像需要做颜色空间转换。使用 cscc.lib 的函数完成转换。在添加 cscc.lib 的过程中，若提示找不到 libc.lib，则忽略该库。

引入头文件 convert.h，在 CmainFrame 类中添加变量。

```
ColorSpaceConversions conv;   //颜色空间转换类
```

修改视频处理回调函数，具体代码如下。

```
LRESULT FAR PASCAL VideoCallbackProc(HWND hWnd, LPVIDEOHDR lpVHdr)
{
 unsigned char *bufi, *buf;
 int type=0;               //编码类型: I、P、B
 int quant=0;              //量化步长
 int declen=0;             //解码长度
 int enclen=0;             //编码长速
 bufi = new unsigned char[lpVHdr->dwBytesUsed+40];    //原图像
 buf = new unsigned char[lpVHdr->dwBytesUsed];        //编码后的码流
 memcpy((void *)(bufi), lpVHdr->lpData, lpVHdr->dwBytesUsed);//获取视频图像
 unsigned char *buf1;
 buf1 = buf;
 if (m_vfwState==ENCDEC) { //使用ICM编/解码器
     //Encode
        buf1 = (unsigned char*)ICSeqCompressFrame(&pc,0,bufi, &IsKeyFrame,&
FrameSize);
        if (bSaveAVI){
               AVIStreamSetFormat(pMainFrame->ps,
                                  pMainFrame->m_Frame++,lpbiTmp,sizeof
                                  (BITMAPINFO));
               AVIStreamWrite(pMainFrame->ps,pMainFrame->m_Frame, 1,
                             (LPBYTE)buf1, lpbiTmp->bmiHeader.biSizeImage,
                             AVIIF_KEYFRAME,NULL,NULL);
        }
     //解码
        ICDecompress(hic2,0,&lpbiTmp->bmiHeader,buf1,&lpbiOut->bmiHeader,
&bufo[40]);
   } else {                      //使用自己的算法：先编码后解码，然后转换到RGB24
        enc_main(bufi,buf, &IsKeyFrame,&type,&quant,&enclen);
                                 //xvid mpeg-4视频编码
        declen = dec_main(buf, bufi, enclen,lpbiIn->bmiHeader.biWidth);
                                 //xvid mpeg-4视频解码
```

```
        pMainFrame->conv.YV12_to_RGB24(bufi,
                                    //解码后的YUV420转换为RGB
                                   bufi+(lpbiIn->bmiHeader.biWidth*lpbiIn->bmiHeader.
biHeight),
                                   bufi+(lpbiIn->bmiHeader.biWidth*lpbiIn->
bmiHeader.biHeight*5/4),
                                   &bufo[40],
                                   lpbiIn->bmiHeader.biWidth,
                                   lpbiIn->bmiHeader.biHeight);
    }
    pMainFrame->GetActiveView()->InvalidateRect(NULL,FALSE);//送显示
    delete bufi;    //释放空间
    delete buf;    //释放空间
    return (LRESULT) TRUE;
   }
```

视频流处理回调函数中，使用自己的处理算法，先进行视频编码然后解码，把解码后的YUV420 格式的数据转换成 RGB24，最后送显示。

关闭 XviD CODEC 编/解码器，销毁编/解码器实例，释放内存。实现代码如下。

```
   void CMainFrame::OnClose()
   {
    //TODO：在此添加消息处理程序代码和/或调用默认值
    capCaptureAbort(m_hWndCap);                              //终止视频捕获
    capDriverDisconnect(m_hWndCap);                          //断开与设备的连接
    Sleep(100);                                              //等待确保前面的操作成功
    capSetCallbackOnError(m_hWndCap,NULL);                   //注销出错回调函数
    capSetCallbackOnStatus(m_hWndCap,NULL);                  //注销状态回调函数
    capSetCallbackOnVideoStream(m_hWndCap,NULL);             //注销视频流处理回调函数
    delete lpbiIn;           //释放输入的位图头结构
    delete lpbiTmp;          //释放临时的位图头结构
    delete lpbiOut;          //释放输出的位图头结构
    if (m_vfwState==ENCDEC){                           //使用了ICM编/解码器
          ICDecompressEnd(hic2);                       //解码结束
          ICClose(hic2);                               //关闭解码器
          ICSeqCompressFrameEnd(&pc);                  //结束序列编码
          ICCompressEnd(hic1);                         //编码结束
          ICClose(hic1);                               //关闭编码器
          AVIStreamClose(ps);                          //关闭AVI文件流
          if(m_pFile != NULL)
                AVIFileRelease(m_pFile);               //关闭AVI文件
    }
    enc_stop();              //销毁用户的编码器
    dec_stop();              //销毁用户的解码器
    Sleep(100);
    CFrameWnd::OnClose();
   }
```

在退出应用程序前，把所有申请的空间、资源等释放或清空，断开 VFW 设备驱动，关闭 ICM 编码器、解码器，关闭 AVI 文件有关操作，销毁用户的编码器和解码器。

10.5 平台应用效果展示

该案例基于 VC2005 单文档的应用程序框架，使用 VFW 采集、预览视频；视频编码、解码，实时显示处理后的图像。主界面如图 10-8 所示。

1）平台应用一效果展示

vfwapp 项目采用 ICM 管理模块的方式，使用系统已注册的编/解码库（插件），对 VFW 采集捕获的视频做编码和解码处理。该应用没有做颜色空间转换，所以在配置视频格式时确定选择 RGB 格式，这样经过编/解码插件处理后，视频图像仍然为 RGB 格式。

首先“初始化 VFW”完成 VFW 设备的准备工作，如图 10-9 所示。然后“配置视频格式”，激活“视频格式”和“视频源”控制页。最后“预览视频”。根据自己的需要，“捕获编/解码”显示解码图像并存盘 AVI 文件。

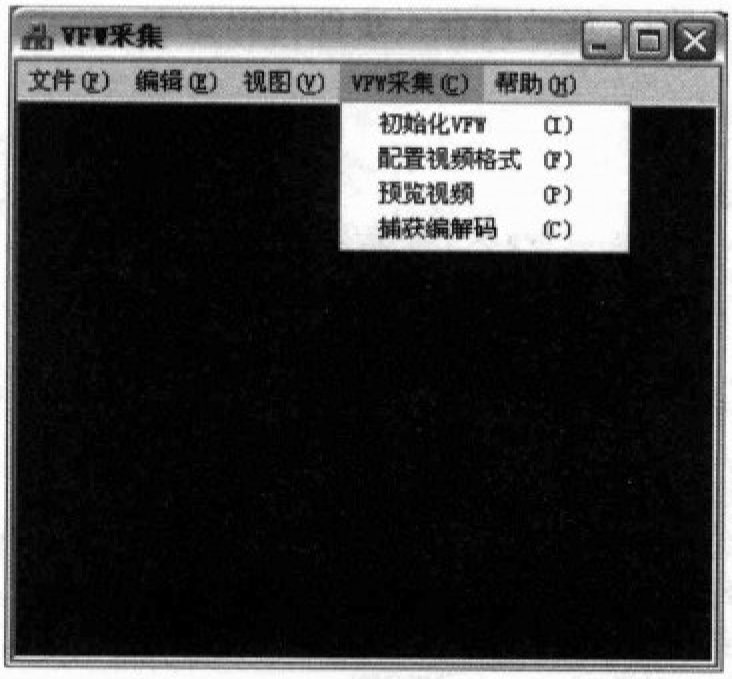


图 10-8 VFW 采集主界面

图 10-9 配置 VFW 设备

ICM 管理的编/解码插件，使用标准接类 COMPVARS 来配置编码器的编码参数。编码器对开发者是“透明”的。

2）平台应用二效果展示

vfwapp_2 项目完成功能基本同 vfwapp，只是没有存盘 AVI 文件功能。该应用主要展示如何对视频流做直接处理，并观看处理后的效果。平台使用自己编译的 XviD CODEC 静态编/解码算法库 libxvidcore.lib 对视频做直接处理。平台应用的编码器参数控制灵活，数据处理流程清晰。注意配置采集的视频格式为 I420，因为 XviD CODEC 算法只支持 I420。

通过修改参数直接观看图像效果，如解码图像的去块效应 deblock。图 10-10 为无 deblock 的解码图像，图 10-11 为有 deblock 功能的解码图像。其中视频分辨率 CIF，码流 80kbps。

通过修改自己的算法参数，实时地浏览 deblock 的效果。有了该平台开发人员就可以增加其他更为复杂的图像处理算法了。

图 10-10 无 deblock 解码图像

图 10-11 有 deblock 解码图像

3）两种平台的对比

对于第一种平台，它是预先安装相应的多媒体编/解码器，使用 VFW 管理已经注册的编/解码器。这种方式对开发者来说，缺点是不能直接访问视频数据，优点是开发者不需要了解视频编码的太多的细节，能够快速构建出基于 VFW 的视频采集、压缩的应用程序。而第二种平台，开发者直接访问数据，除了可以做编码外，还可以进行其他处理：识别、滤波或增强等。缺点是开发者需要掌握处理算法，编程较复杂，优点是应用程序功能强大，控制数据更加灵活。

10.6 本章小结

VFW 是 Windows 自带的视频捕获、播放 SDK 软件包，它是针对早期捕获设备的驱动支持。它由 6 个动态链接库或驱动模块构成，使用 VFW 可以快速构建视频应用程序，并能对视频源格式和图像参数进行动态设置。案例在基于 VFW 的采集和预览功能下，以两个实例应用展示平台的通用性：使用 VFW 的 ICM 控制系统注册的视频压缩器完成视频编码压缩、AVI 存储和解码图像显示；使用 XviD CODEC 算法直接控制视频流，展现算法的可视效果，便于用户对采集的数据做其他算法处理。每种平台有不同的应用领域和开发特点，后者更专业，前者更简单。尽管 DirectShow SDK 已经被广泛地应用到视频的采集、预览任务中，而早期的 VFW 技术鉴于其原理相对简单、网络或书籍等开发资源丰富，目前仍然是视频采集任务的主要实现方法。

[illegible]

[illegible]

第4篇

编/解码技术

本篇是高级应用篇，编/解码是音/视频处理的重要方法。首先概述了数字音/视频的编/解码方法，回顾了当前的编/解码技术标准及应用领域和技术指标，总结了算法开源资源，展望了编/解码产业。接着介绍了多媒体计算机的MMX/SSE多媒体汇编指令，剖析了指令特点，通过案例详细介绍MMX和SSE进行算法优化的过程并展示技术效果，使用汇编指令优化视频编/解码算法，并列举了开发常见问题。然后展示使用DSP（DM642）实现视频编/解码，包括CCS的安装、开发过程，使用C语言、线性汇编来设计、优化算法的思路，并以创建库工程和应用工程为对象分析开发常见问题。最后以3个实用工程案例详细介绍了视频算法开发和系统应用，包括开源代码XviD CODEC实现MPEG-4视频编/解码算法、开源代码X264实现H.264/AVC视频编码和DVR视频监控系统设计等。这部分案例读者可以直接参考完成开发视频应用，或基于这些工程设计出用户自己的程序。

- 第11章　纵览音/视频编/解码技术
- 第12章　使用MMX/SSE2进行多媒体编程
- 第13章　用DM642实现视频编/解码技术
- 第14章　XviD CODEC实现MPEG-4编/解码
- 第15章　X264实现H.264/AVC视频编码
- 第16章　打造自己的DVR监控系统

[illegible]

[illegible]

第 11 章

纵览音/视频编/解码技术

数据的数字化是当今主流数据处理的前提，音/视频数据是人类从外界获取信息的绝对信息源，约占总信息量的 94%。数字音/视频的处理主要包括获取、滤波、编/解码、识别、增强等，鉴于音/视频数据的固有特性，编/解码处理又成为其重要处理方式。并且音/视频编/解码已经成为了一个产业，如 VCD、DVD、MP3、MP4 或 DC、DV 视频点播（VOD）、音/视频监控（DVR）、数字电视等音/视频的应用中都需要对音频、视频或图像等数据源做编码压缩或解码等处理。音/视频编码技术已经形成了国际标准，如 JPEG、JPEG 2000、MPEG-1/2/4、H.263/H.264、国产 AVS 标准。制定音/视频标准的国际组织主要有 ISO/IEC 和 ITU.T。

音/视频、图像的编/解码算法国际标准定义了数据流的格式和语法，没有规定软件的实现过程。已有众多的公司企业、研究机构参与算法的研制和工程实现。开源算法考虑了工程应用的运行效率和平台特点，本章主要介绍音/视频、图像编/解码的技术原理。

11.1 数字音/视频基础

我们已经进入了一个信息时代，人类从外界获取的信息有83%来自视觉，11%来自听觉。所以音频、语音和视频、图像等信号是信息时代重要的处理数据源，这些信号的数字化也是必然趋势。我国也已经计划到2015年全面施行数字电视普及化。数字化音/视频产品已经渗透到我们日常生活及工作的方方面面，例如MP3、MP4、DVD、DC、DV、多媒体计算机、影视及投影设备等众多数字化影音产品。

数字化包括广义数字化和狭义数字化。广义数字化即从数据源的获取、传输、处理和存储等所有过程均为数字化；狭义数字化指处理的数据为二进制的数字化数据，如光盘、计算机文件等。我们要研究的数字化是狭义数字化，如数字音频数据、视频数据。音频的数字化指在计算机中存储的WAV、PCM、MP3等格式的语音文件，视频数字化指的是视频数据是像素点，以RGB三基色或YUV 3个分量形式存储在计算机的硬盘中，或者以某种压缩格式形成的AVI、MPG、ASF等影音文件。音/视频的数字化是对其做处理的基础，我们要处理的对象就是这些数字音/视频数据：对音/视频做变换、量化或编码等算法处理，对数字图像做滤波、增强，对运动物体做识别、跟踪，对语音做识别、融合等。

1．数字音频

数字化的音频是利用AD转换设备将模拟语音、音频信号按照一定的采样频率采集、量化成的数字音频信号。语音常用采样频率8kHz，音频常用采样频率44.1kHz或更高。在相同采样频率条件下，语音、音频码流越高，保真度越高，听起来越舒服。如我们常见的MP3，多数是128Kbps或192Kbps，44.1kHz采样，不过这种格式的音频听起来也逼真。

一般我们常见的没有做编码的数字音频数据格式是PCM，16位样点，8、32、44.1kHz采样频率。CD格式的音频就是没有采集的原始语音，几乎没有任何信息损失，但码流较大，如128Kbps，一个CD只能存储几首歌曲。常见的MP3，几MB的文件可以播放4～5分钟，并且效果也非常好，专业的MP3播放器效果处理得更是逼真。

在通信中的音频、语音码流要求尽可能的低，如8Kbps，甚至更低。如ITU-T的G.72X语音系列编码标注，码流8/16/24/32/64Kbps，适合通信中的窄带传输要求。

语音通道的多少对听觉效果也有很大的影响，如5.1声道、高保真、杜比环绕立体声等都是多声道，码流数据量也随之上升。

2．数字视频

我们谈及的数字视频更多地指视频算法待处理的视频数据，如RGB、YUV等格式。视频的来源一方面是通过模拟摄像头，经过视频采集芯片（视频A/D：菲利普SAA 7121）按照预定格式采集。另一方面是通过USB摄像头，如QQ聊天常用的小摄像头，计算机采集到的数据就是数字视频。数字视频算法处理的数据多数是YUV格式的，因此采集设备也绝大部分支持该格式。

另外，视频的数字化对于不同格式的 RGB 或 YUV 来说，每个分量的比特数或长度也不尽相同。如分量有 12bits、10bits 或 8bits 的区别，对于消费类电子视频产品，8bits 分量占主流。而 YUV 又分为 YUV 420、YUV 422，ITU-R 的 BT.601 和 BT.656 是 YUV 422 格式。视频算法处理的视频分量多数是 YUV 420，即使输入其他格式，编码前也要做格式转换。视频算法解码后仍然是 YUV 420，如果要显示到电视等设备，则一般要再转换成 YUV 422，即 BT.656 格式。当然也有直接处理 RGB 或 YUV 444 格式的算法，如静态图像编码标准 JPEG 2000。

针对 YUV 420 的视频算法视频源格式，根据其分辨率大小又分为多种，如表 11-1 所示。常用的有 CIF（352 像素×288 像素 PAL，公共中间格式）、D1（704 像素×576 像素 PAL，标清 D1）。

表 11-1 ITU-R 建议的图像格式

图像格式	亮度分辨率	色度分辨率
SQCIF	128 像素×96 像素	64 像素×48 像素
QCIF	176 像素×144 像素	88 像素×72 像素
CIF	352 像素×288 像素	172 像素×144 像素
4CIF/D1	704 像素×576 像素	352 像素×288 像素
16CIF	1408 像素×1152 像素	704 像素×576 像素

3. 压缩算法

首先来分析表 11-2 中的数据，音/视频数据的海量是显然的。另外理论已经证明原始数据本身有冗余（如时域和空域数据的强相关性）；第三方面人的视觉和听觉有其生理特点（对语音高频不敏感、对图像色度不如亮度敏感），这就带来数据压缩的必要性和可行性。

表 11-2 未压缩信源的大致比特率

未压缩信源	比特率
电话（200～3400Hz ）	8000 样本数/秒×12 比特/样本=96Kbps
宽带语音（50～7000Hz）	16000 样本数/秒×14 比特/样本=224Kbps
宽带音频（20～20000Hz）	44100 样本数/秒×2 信道×16 比特/样本=1.412Mbps
图像	512×512 像素色彩图像×24 比特/像素=6.3 兆比特/图像
视频	640×480 像素色彩图像×24 比特/像素 × 30 图像/秒=221Mbps

语音、音频压缩算法常用的有 G.72X（711、723、726、729 等）、MP3、WMA。静态图像编码标准有 JPEG、JPEG 2000、GIF、TIFF 等。动态视频编码标准有 MPEG-X（1、2、4）、H26X（1、3、4）、AVS（Advanced Audio-Video Coding-Decoding Standard）等。

ITU 和 ISO 为制订标准和协议的国际组织，他们有自己制订的标准同时也有联合发布的标准，如 JVT（Joint Video Team，联合视频组）发布的 MPEG-4 AVC/H.264 就是同一个协议在不同的机构中的不同的代号及俗称。常见的图像、视频国际标准典型应用如表 11-3 所示，其中 AVS 为中国制订的完全知识产权的先进影音编/解码标准。

表 11-3　音/视频、图像压缩国际标准

标准号	俗称	适用信源	典型应用
ITU-T T .82 | ISO/IEC 11544	JBIG-1	二值图像、图形	G4 传真机、计算机图形
ISO/IEC 14492	JBIG-2	二值图像、图形	传真、www 图形库、PDA 等
ITU-T T .81 | ISO/IEC 10918	JPEG	连续色调静止图像	图像库、传真、彩色印刷、数码相机
ITU-T T .87 | ISO/IEC 14495	JPEG-LS	连续色调静止图像	医学、遥感图像资料的无损/近似无损压缩
ISO/IEC 15444	JPEG2000	连续色调静止图像	各种图形、图像（含计算机生成的）
ITU-T G.726、G.728 和 G.729	G.72X	语音	数字通信和电话录音
ITU-T H.261 建议	P×64	活动图像	ISDN 上的会议电视/可视电话
ITU-T H.263 建议	H.263	活动图像	PSTN 上的会议电视/可视电话
ISO/IEC 11172	MPEG-1	活动图像及伴音	VCD、DAB、多媒体、VOD 等
ITU-TH.262|ISO/IEC 13818-2	MPEG-2 视频	高质量活动图像	SVCD/DVD、VOD/MOD、多媒体视频游戏、DVB、DTV/HDTV 等
ISO/IEC 13818-3	MPEG-2 音频	高质量多声道声音	DAT、DCC、DAB 等级数字视频伴音
ISO/IEC 14496	MPEG-4	多媒体音像数据	www 上的视频、音频扩展
ITU-T H.264 建议（MPEG-4 part 10）	AVC	各种活动图像	H．261/263 和 MPEG-1/2 应用的替代
先进音、视频编码系列标准	AVS 标准	活动图像和音频	广播电视、音像产品和多媒体通信等

11.2　音/视频编/解码及方法

音/视频数据的编码方法有很多种，不同的组织制定针对不同应用的算法标准。音频编码的算法发展较慢，视频编码标准发展迅速。特别是视频编码压缩算法，从较早的 MPEG-1/H.261 到现在的 MPEG-4/H.264 或 AVS，编码算法的码流更低、图像质量更好，随之算法的运算量也倍增，如 H.264 编码运算量是 H.263 的 3 倍，而效果确实提高很多：如图像 PSNR 指标比 MPEG-4 提高 2dB，比 H263 提高约 3dB。实际工程应用的 MPEG-4 与 H.264 都是基于块的混合编码技术，即宏块的变换、量化和编码方法。

11.2.1　音频编码方法

语音压缩编码是利用语音信号数字处理，以有限的比特数来表示语音信息，从而达到语音压缩的目的。ISO 制订了 VCD、DVD 等存储用途的音频编码标准如 MPEG、AC-3、AAC 等，针对影音存储，注重回放效果。ITU 制订了 G.7xx 系列针对多媒体通信中的语音编码标准，注重低码流。典型语音压缩标准的参数比较如表 11-4 所示。

表 11-4 ITU-G 系列语音压缩标准参数比较

算法	类型	码率（Kbit/s）	算法延时（ms）
G.711	A-Law / μ -Law	64	0
G.722	SB-ADPCM	64/56/48	0
G.723.1	MP-MLQ/ACELP	6.3/5.3	37.5
G.726	ADPCM	16/24/32/40	0
G.727	Embedded ADPCM	16/24/32/40	0
G.728	LD-CELP	16	<2
G.729	CS-ACELP	8	15

1．语音编码技术

语音编码即电话质量的语音信号的编码，语音频率的范围规定为 300Hz～3.4kHz，采用标准的脉冲编码调制（PCM），当采样频率为 8kHz 进行 8bit 量化时，所得数据速率为 64Kbit/s，即一个数字话路的语音速率。1972 年 CCITT（即现在的 ITU-T）制定了 PCM 标准 G.711，速率为 64Kbit/s，采用非线性量化 μ 律或 A 律，其质量相当于 12bit 线性量化。1984 年 CCITT 公布了自适应差分脉冲编码调制（ADPCM）标准 G.721，速率为 32Kbit/s。这一技术是对信号和它的预测值的差分信号进行量化，同时再根据邻近差分信号的特性自适应改变量化参数，从而提高压缩比，又能保持一定信号质量。因此 ADPCM 对中等电话质量要求的信号能进行高效编码，而且可以在调幅广播和交互式激光唱盘音频信号压缩中应用。

为了适应低速率语音通信的要求，必须采用参数编码或混合编码技术，如线性预测编码（LPC）、矢量量化（VQ），以及其他的综合分析技术。其中较为典型的码本激励线性预测编码（CELP）实际上是一个闭环 LPC 系统，由输入语音信号确定最佳参数，再根据某种最小误差准则从码本中找出最佳激励码本矢量。CELP 具有较强的抗干扰能力，在 4～16Kbit/s 传输速率下即可获得较高质量的语音信号。

1992 年 CCITT 制定了短时延码本激励线性预测编码（LD-CELP）的标准 G.728，速率为 16Kbit/s，其质量与 32Kbit/s 的 G.721 标准基本相当。1988 年欧洲数字移动特别工作组制定了采用长时延线性预测规则码本激励（RPE-LTP）标准 GSM，速率为 13Kbit/s。1989 年美国采用矢量和激励线性预测技术（VSELP），制定了数字移动通信语音标准 CTIA，速率为 8Kbit/s。为了适应保密通信的要求，美国国家安全局（NSA）分别于 1982 年和 1989 年制定了基于 LPC、速率为 2.4bit/s 和基于 CELP、速率为 4.8Kbit/s 的编码方案。

语音编/解码算法根据处理的数据内容大体上可分为 3 类：波形编码、参数编码和混合编码。

1）波形编码（Wave Coding）

语音信号的波形编码力图使重建语音波形保持原语音信号的波形形状，也即失真要最小。这类编码器通常将语音信号作为一般的波形信号来处理，它具有适应能力强、话音质量好等优点，但所需的编码速率高。波形编码比较简单，编码前用采样定理对模拟语音信号进行量化，然后进行幅度量化，再进行二进制编码。解码器作数/模变换后再由低通滤波器恢复出原始的模拟语音波形，这就是最简单的脉冲编码调制（PCM），也称为线性 PCM。自适应

增量调制（ADM）、自适应差分编码（ADPCM）、自适应变换编码（ATC）等都属于这类编码器。波形编码的方法简单，数码率较高，在 64～32Kbit/s 之间音质优良，当数码率低于 32Kbit/s 的时候音质明显降低。

2）参数编码（Parametric Coding）

参数编码又称为声码化编码、模型编码，它通过对语音信号特征参数的提取及编码，力图使重建语音信号具有尽可能高的可懂性，即保持原始语音的语意，而重建语音信号的波形和原始语音信号的波形可能会有很大的差别。这类编码器编码后的码率可以做得很低，如 1.2Kbit/s 和 2.4Kbit/s，但是合成语音质量较差，复杂度比较高，对环境要求较高。典型的参数编码器有 LPC-10（Linear Predictive Coding，线性预测编码）、LPC-10E，当然，G.729、G.723.1 及 CELP（FS-1016）等码本激励语音编码器都离不开参数编码。

3）混合编码

混合编码算法结合了上述两种方法的优点，既有波形逼近的环节，又充分利用语音产生模型和人类听觉系统的特性，因此可以在中等速率（4～16Kbit/s）下提供高质量（即主观测试为 good 等级）的输出语音。典型的编码算法有矢量和激励线性预测（VSEI. P）、多脉冲线性预测编码（MP-LPC）、规则脉冲激励线性预测编码（RPELPC）和码激励线性预测（Code Excited Linear Predictive，CELP）声码器等。

在上述 3 种语音编码中，波形编码原理简单，压缩倍数低，语音质量好；参数编码原理复杂，压缩倍数高，语音质量相对较差；混合编码压缩倍数中等，语音质量较好。语音相关压缩标准及功能描述如下。

- G.723：一种 ITU-T 推荐标准，用于传输速率在 5.3～6.4Kbps 之间的多媒体通信传输的双速率语音编码器。
- H.221：ITU-T 的 H.320 推荐标准的框架部分，被正式称为“视听电话服务中 64～1920Kbps 通道的框架结构”，该推荐标准叙述了能让编码器和译码器在时间上同步的同步操作。
- H.222：ITU-T 推荐标准，规定了运动图片及相关音频信息的通用编码。
- H.223：ITU-T 的 H.324 标准的一部分，一个控制/复用协议，通常被叫做“用于低位率多媒体通信的复用协议”。
- H.233：一种复用推荐标准，是 ITU-T 视频互操作推荐标准协议族的一部分。该推荐标准规定音视信息的单个画面如何在数字频道中进行复用。
- H.231：附加于 ITU-T 的 H.320 协议族的推荐标准，规定多点控制单元，用于多点会议中将 3 个或 3 个以上遵从 H.320 的编/解码器（Codec）桥接在一起。
- H.242：ITU-T 的 H.320 协议族中视频互操作推荐标准部分，它规定了建立一个音频会话和在通信终止后结束该会话的协议。
- H.245：ITU-T 的 H.323 和 H.324 协议族部分，定义多媒体终端之间的通信控制。
- H.320：一个 ITU-T 标准，它包含了大量的单个推荐标准，如编码、组帧、信号及建立连接（H.221、H.230、H.321、H.242 及 H.261）等。应用于点对点和多点可视会议会话，且包含 G.711、G.722 和 G.728 这 3 种音频算法。
- H.323：H.323 将 H.320 扩展到了内联网、外联网和互联网的包交换网络中，例如

以太网、令牌环和其他一些可能不保证 QoS 的网。它也规定了 ATM 包括 ATM QoS 上可视会议过程，它支持点对点和多点操作。

- H.324：一个 ITU-T 标准。它在模拟电话线（POTS）上提供了点对点的数据、视频和音频会议。H.324 协议族包括 H.223（一种多路复用协议）、H.245（一种控制协议）、T.120（一套音频图像协议）和 V.34（一种调制解调器规范）。
- T.120：ITU-T 的“多媒体数据传输协议”，一种数据共享/数据会议规范，使用户能通过任何 H.32x 可视会议共享文件。

2. 音频编码技术

音频即信号频率范围是 20Hz～20kHz，采用 48kHz、44.1kHz、32kHz 等较高采样频率，16bit 量化进行数字化转换，其数据速率每声道可高达 705Kbit/s。一般语音信号的动态范围和频响比较小，采用 8kHz 采样，每样值用 8bits 表示，现在的语音压缩技术可把码率从原来的 64Kbps 压缩到 4Kbps 左右。但多媒体通信中的声音要比语音复杂得多，它的动态范围可达 100dB，频响范围可达 20Hz～20kHz。因此，声音数字化后的信息量也非常大，例如把六声道环绕立体声数字化，按每声道取样频率 48kHz，每样值 18bits 表示，则数字化后的数据码率为 6×48kHz×18bits＝5.184Mbit/s，即使是两声道立体声，数字化后码率也达到 1.5Mbps 左右，而电视图像信号数字压缩后码率大约为 1.5～10Mbps，因此，相对而言声音未经数字压缩的码率就太高了，为了更有效地利用宝贵的信道资源，必须对声音进行数字压缩编码。

音频的编码技术主要有自适应变换编码（频域编码）、心理声学模型和熵编码等技术。

1）自适应变换编码

利用正交变换把时域音频信号变换到另一个域，由于是相关的结果，变换域系数的能量集中在一个较小的范围，所以对变换域系数最佳量化后，可以实现码率的压缩。理论上的最佳量化很难达到，通常采用自适应比特分配和自适应量化技术来对频域数据进行量化。在 MPEGlayer3 和 AAC 标准及 DolbyAC-3 标准中都使用了改进的余弦变换（MDCT）；在 ITUG.722.1 标准中则用的是重叠调制变换（MLT），本质上都是余弦变换（DCT）的改进。

2）心理声学模型

其基本思想是对信息量加以压缩，同时使失真尽可能不被觉察出来，利用人耳的掩蔽效应就可以达到此目的，即较弱的声音会被同时存在的较强的声音所掩盖，使人耳无法听到。在音频压缩编码中利用掩蔽效应，就可以通过给不同频率处的信号分量分配以不同的量化比特数的方法来控制量化噪声，使得噪声的能量低于掩蔽阈值，从而使人耳感觉不到量化过程的存在。在 MPEGlayer2、MPEGlayer3 和 AAC 标准及 AC-3 标准中都采用了心理声学模型，在目前的高质量音频标准中，心理声学模型是一个最有效的算法模型。

3）熵编码

根据信息论的原理，可以找到最佳数据压缩编码的方法，数据压缩的理论极限是信息熵。如果要求在编码过程中不丢失信息量，即要求保存信息熵，这种信息保持编码叫熵编码，它是根据信息出现概率的分布特性而进行的，是一种无损数据压缩编码。常用的有霍夫曼编码和算术编码。在 MPEGlayer1、MPEGlayer2、MPEGlayer3 和 AAC 标准及 ITUG.722.1 标准中都使用了霍夫曼编码；在MPEG4 BSAC 工具中则使用了效率更高的算术编码。

为了统一制定音频、视频编码方案，ISO/IEC 标准组织成立了 ISO/IES JTC1/SC29/WG11，

即 MPEG（活动图像专家组）。该小组负责比较和评估几种低码速率数字声音编码技术，以产生一套国际标准，用于活动图像、相关声音信息及其结合，和用数字存储媒体（DSM）存储与重现。MPEG 针对的 DSM 包括 CD-ROM、DAT、磁光盘和电脑磁盘。基于 MPEG 的压缩技术还将用于多种通信信道，如 ISDN、局域网和广播。

“低于 1.5Mbit/s 的用于数字存储媒体的活动图像和相关声音之国际标准 ISO/IEC”（MPEG-1）于 1992 年 11 月完成。其中 ISO ll172-3 作为“MPEG 音频”标准，成为国际上公认的高保真立体声音频压缩标准，一般称为“MPEG-1 音频”。MPEG-1 音频第一和第二层次编码是将输入音频信号进行采样频率为 48KHz、44.1KHz、32KHz 的采样，经滤波器组将其分为 32 个子带，同时利用人耳屏蔽效应，根据音频信号的性质计算各频率分量的人耳屏蔽门限，选择各子带的量化参数，获得高的压缩比。MPEG 第三层次是在上述处理后再引入辅助子带，非均匀量化和熵编码技术，再进一步提高压缩比。MPEG 音频压缩技术的数据速率为每声道 32～448Kbit/s，适合于 CD 光盘应用。

MPEG-2 也定义了音频标准，由两部分组成，即 MPEG-2 音频（Audio，ISO/IEC 13818-3）和 MPEG-2 AAC（先进的音频编码，ISO/IEC 13818-3）。MPEG-2 音频编码标准是对 MPEG-1 后向兼容的、支持二至五声道的后继版本。主要考虑到高质量的 5+1 声道、低比特率和后向兼容性，以保证现存的两声道解码器能从 5+1 个多声道信号中解出相应的立体声。MPEG-2 AAC 除后向兼容 MPEG-1 音频外，还有非后向兼容的音频标准。

MPEG-4 Audio 标准 AAC（ISO/IEC 14496-3）可集成从话音到高质量的多通道声音，从自然声音到合成声音，编码方法还包括参数编码、码激励线性预测编码、时间/频率（Time/Frequency，T/F）编码、结构化声音（Structured Audio，SA）编码、文语转换（Text-To-Speech，TTS）的合成声音和 MIDI 合成声音等。

MPEG-7 Audio 标准（ISO/IEC 15938-3）主要提供了音频描述工具，用于检索语音内容或文件。

11.2.2 视频编码方法

视频图像由于其自身固有的特性即空间的相邻像素的强相似、时域的相邻帧之间的强相关、视频数据量巨大和人眼视觉的色度自动掩蔽效应，为视频图像数据实施编码压缩提供了理论依据和实践依据。空间像素的相似为图像帧内编码提供了预测技术，即对相邻宏块的预测差值做编码；时域的相邻帧的相关为图像帧间编码提供了帧间预测、运动估计技术，即对当前宏块与最佳匹配宏块做差，而对残差做编码；视频编码充分利用数学知识，把处理的图像由时域转换到频域，在频率域内做量化规格化，利用计算机知识的统计编码、信息学科中的熵编码完成视频编码。

标准的视频编码算法从最早的 MPEG-1、H.261 到 MPEG-2、H.263，再到现在的 MPEG-4、H.264、AVS 等，码流逐渐降低、图像质量逐渐提高、算法运算量逐渐增大。这些算法是基于像素块的传统视频编码方法，虽然 MPEG-4 的高档级中提供了基于对象的视频编码，但由于视频对象的分割一直困扰着编码人员，所以对象视频编码实际没有较大应用。传统的视频编码是基于像素块的混合编码框架，包括预测、变换、量化和编码。不同的算法标准差别主要在这些处理模块上的区别，如帧内/帧间预测的方法、传统 DCT 与整数 DCT、普通 VLC 与 CAVLC/CABAC 等。

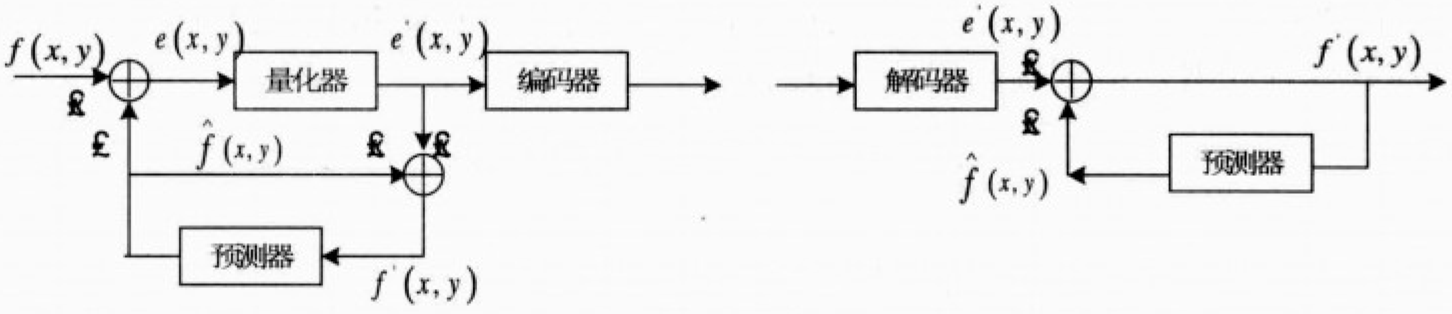
帧间编码是视频序列的主要编码方法，而帧间编码中，最佳匹配块的搜索即运动估计是关键。最佳匹配块准则 SAD（Sum of Absolute Difference）计算，整数像素、半像素、1/4 像素的运动补偿插值等是运动估计中的主要运算量，运动估计在视频编码中约占60%的计算量。

1. 预测编码

预测编码是视频编码的最主要的编码方法，简单而有效。理论和实践都已经证明，图像的像素点与帧内的上、下、左、右等位置的像素点，它们的值变化很小，相关性极强，即图像存在空间冗余；视频帧是一序列活动图像，每秒有多帧图像（实时则需要有 25f/s），当前图像点或图像块与前后帧的对应位置或附近的图像点或块非常相似，相关性极强，即活动序列图像存在时间冗余。预测编码就是减少这些空间冗余和时间冗余，分别对应帧内编码和帧间编码。

1）帧内预测编码

帧内预测编码是对图像帧数据本身做编/解码，如静态图像编码、I帧（或关键帧 key frame）编码都是帧内编码。由于空间像素值不会突变，而是缓慢变化、甚至相同，所以我们在处理当前块时，可以用先前已编码的图像做参考，则零值或较小的值很多，达到了编码的目的。帧内编码的优点是错误少、图像清晰，缺点是压缩比小。在视频序列编码中，帧内是有固定间隔的，如 25f/s×10s=250 帧，即每隔 10 秒出现一个 I 帧帧内编码。图 11-1 是图像预测编码和解码的原理图。



（a）预测编码　　（b）预测解码

图 11-1　图像帧内预测编/解码原理图

图 11-1 中的 $f(x, y)$ 为当前像素值，(x, y) 为像素点水平和垂直坐标。其预测值 $\hat{f}(x, y)$ 可以由当前位置 (x, y) 的周围像素预测得到。

$$\hat{f}(x,y)=\sum_{(k,l)\in Z}\sum \alpha_{k,l} f(x-k,y-l) \qquad (11\text{-}1)$$

$a_{k,l}$ 为预测系数，Z 为预测区域，k、l 分别为对当前点进行预测的像素点的水平和垂直位置坐标值。$e(x, y)$是当前像素值和预测值的差值，差值经过量化、编码形成最后的码流。用户可以发现量化器置于预测环路之内，量化误差被反馈回环路，从而使残存的量化误差减小。

上面是帧内预测编码的原理，但是视频编码的预测是对变换、量化后的系数做预测的，并且根据不同的视频压缩算法标准，预测的方式也有多种。如 MPEG-1/2/4 标准的帧内预测是利用当前块的上方、左方、斜上方等已编码块的第一行或第一列来预测当前块的第一行和第一列的，同时除去第一个直流分量；而 H.264 的帧内预测则根据块的大小进行预测，16×16 亮度块有 4 种预测模式，4×4 亮度块有 9 种预测模式，8×8 亮度块有 4 种预测模式。预测值

和当前值相减，得到预测残差，然后代替当前系数值，从而增加零值和较小值的概率，达到一定的压缩效果。但是，如果图像细节丰富，则帧内预测编码效率往往并不高。

2）帧间预测编码

帧间预测编码是视频编码的主要方法，帧间编码即P帧前向预测编码、B帧双向预测编码。帧间的预测不是在当前图像帧内作预测，而是在已编码重建帧的对应位置或其周围作预测、搜索。因为帧间预测是图像在时间先后上的视频帧处理，所以又非常像一个物体在运动而体现在各个视频帧中。帧间预测就是在一定窗口内搜索到最佳匹配宏块，然后这两个宏块相减得到预测残差。

由于物体的运动轨迹并不是整数像素大小，为了更精确地描述物体的运动轨迹，视频算法提出了半像素、1/4像素、1/8像素的运动估计，此时得到的匹配块和运动矢量就相对更准确。另外如果物体快速运动，则在一定的窗口内无法搜集到最佳匹配块和最真实运动矢量。但是，算法如何在整个图像参考重建帧内搜索，则计算量剧增，从而算法不实用。这里介绍前向预测、1/2 像素差值补偿、菱形运动估计算法的帧间预测，其他方法或更高分数精度像素、复杂运动估计算法可参阅相关资料。

① 帧间预测基本原理

基本原理如图11-2所示，当前帧图像$f_t(x, y)$与预测图像$\hat{f}_t(x, y)$相减后的帧误差$e_t(x, y)$，经量化器量化后输出$e_t'(x, y)$，输出进行熵编码。预测图像$\hat{f}_t(x, y)$与$e_t'(x, y)$相加，得$f_t'(x, y)$，当不计量化失真时，$f_t'(x, y)$即当前的$f_t(x, y)$。把当前帧$f_t'(x, y)$与帧存储器输出的前一帧$f_{t-1}(x, y)$（也称重建参考帧）同时输入运动参数估值器，经搜索、比较得到运动矢量MV。此MV输入运动补偿预测器，得到预测图像$\hat{f}_t(x, y)$。

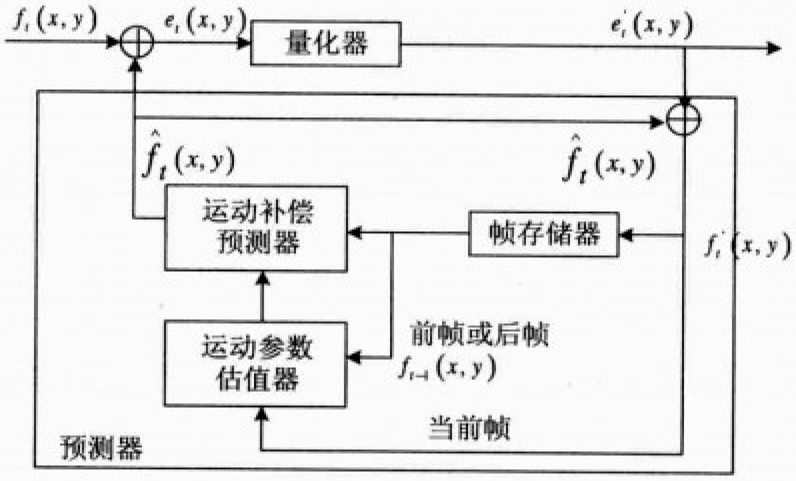


图11-2 前向预测编码

预测图像$\hat{f}_t(x, y)$不可能完全等同于当前图像$f_t(x, y)$，无论预测得如何精确，总存在帧误差$e_t(x, y)$。由上述可知，利用上一帧的图像经运动矢量位移作为预测值的方法称为单向预测。这时：

$$\hat{f}_t(x,y)=f_t(x+i, y+j) \tag{11-2}$$

其中，(i, j)即为当前宏块的运动矢量。预测图像与当前图像对应相减得到残差，对残差$e_t'(x, y)$作熵编码，同时编码运动矢量。

② 重建图像的插值

连续运动的物体，其轨迹是连续的，而像素是离散的。所以只用整数像素来表示运动物体显然是不准确的，所以有非整数像素运动轨迹之称。对于前向预测编码的参考帧只有重建

参考帧，所以就需要在此参考帧的基础上插值出多帧非整数像素运动轨迹的参考帧。如 1/2 像素运动轨迹有 4 个参考帧：重建参考帧、重建参考帧的水平插值参考帧、重建参考帧的垂直参考帧、重建参考帧的斜角参考帧。如 MPEG-4 算法中 1/2 像素插值用到的底层函数 Interpolate16x16_halfpel_h、interpolate16x16_halfpel_v、interpolate16x16_halfpel_hv 分别得到 3 个方向的插值参考帧。更高精度的插值则是在前面较低精度基础上继续在各个方向上插值。

③ 运动估计算法

在以上参考帧中对当前宏块搜索最佳匹配宏块，估计出最佳运动矢量大小。这里介绍的运动估计算法是基于块匹配的运动估值算法，即对当前编码宏块寻找最佳匹配图像宏块，同时就找到了最佳运动矢量 mv（dx,dy）。运动估计的方法有多种，搜索的范围、大小也有多种；另外这里的最佳也是有限范围内的最佳，并且最佳准则也有多种。所以运动估计要解决两个问题：匹配准则和搜索算法。

- 匹配准则：常用的块匹配判断准则有 MAD（平均绝对差）、SAD（差的绝对值的和），公式分别见 11-3 和 11-4。

$$MAD = \frac{1}{I \times J} \sum_{|i| \le \frac{I}{2}} \sum_{|j| \le \frac{J}{2}} \left| f(i,j) - g(i-d_x, j-d_y) \right|, I = J = 16 \tag{11-3}$$

$$SAD = I \times J \times MAD \tag{11-4}$$

由于 SAD 只有乘加没有除法，所以 SAD 准则更为常用。SAD 值越小，表明这两个块越相似；MAD 值的意义判断同 SAD。

- 搜索算法：对当前预测编码图像的宏块在参考帧中，在一定窗口内，搜索判断，根据准则 SAD 在窗口限制范围内搜索最佳匹配宏块 $f(x,y)$ 和运动矢量 mv(dx,dy)，运动估计原理如图 11-3 所示，根据搜索的路径不同有多种运动估计算法。

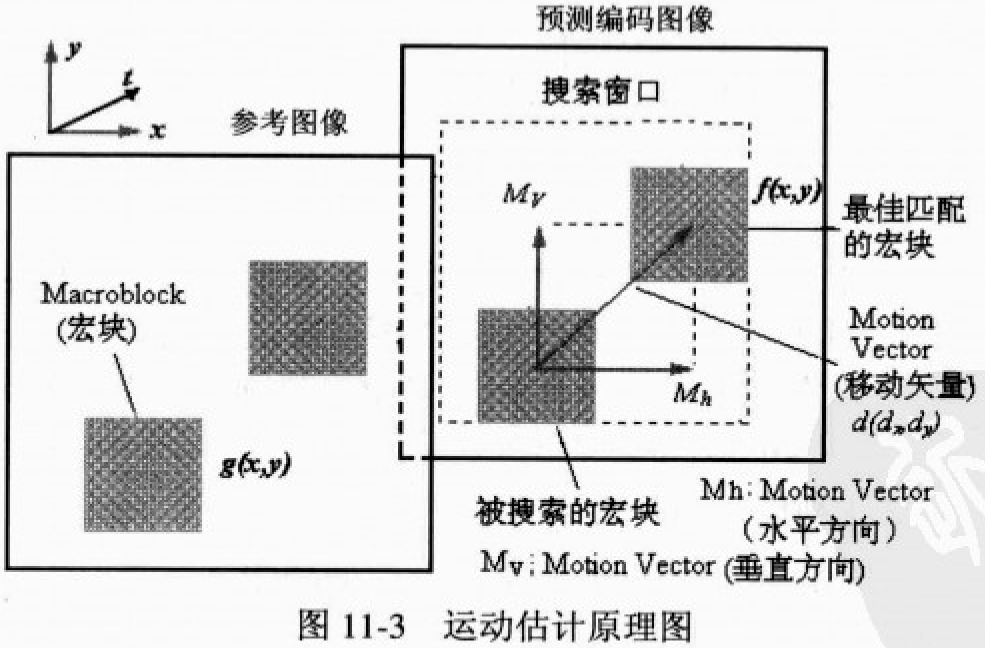


图 11-3　运动估计原理图

最简单的运动估计算法有三步搜索法、对数搜索法、四步搜索、菱形搜索、大/小钻石搜索、六角形搜索等。

三步搜索算法如图 11-4 所示，每一步搜索 9 个位置点，搜索步长为 3。对每一个位置计算 MAD/SAD 值，确定最小的失真方向，在最小失真方向上搜索区域减少一个搜索步长，下一步搜索新的 9 个位置点。直到搜索步长为 1，此时的宏块即为用户认为的最佳匹配块，并记录当前运动矢量。

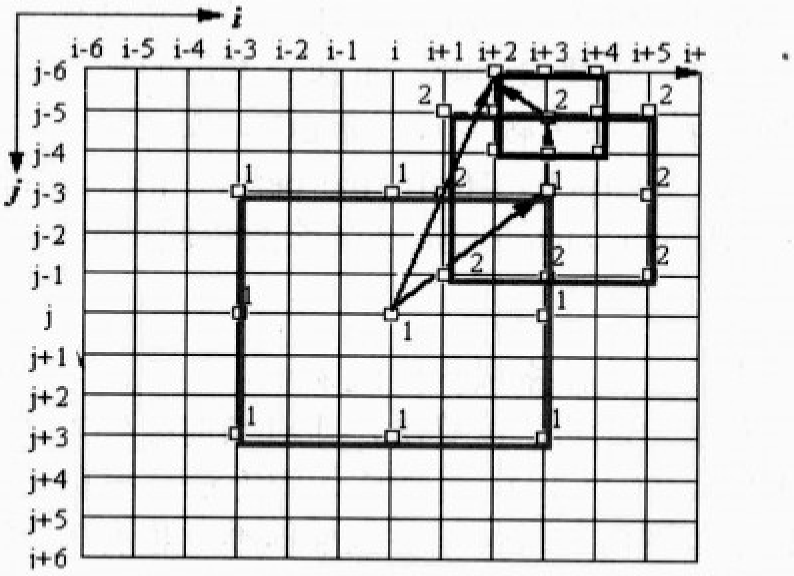


图 11-4　三步搜索算法示意图

对数搜索法如图 11-5 所示，每一步搜索由区域中心位置和与中心相连的 4 个边缘点组成，连续搜索直到寻找的区域成为 3×3 矩形区域为止。最后一步计算 3×3 矩形区域 9 个位置点，MAD/SAD 最小值对应的宏块就是要寻找的最佳匹配块。

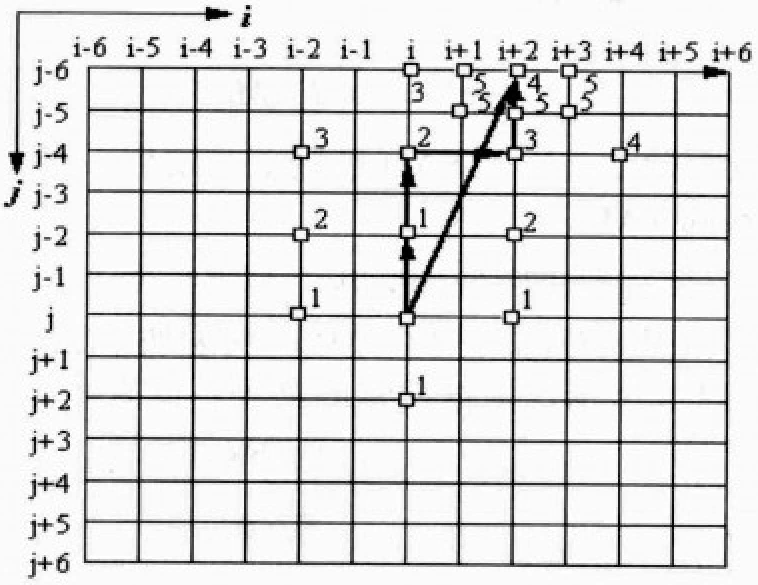


图 11-5　对数搜索方法

由于实际物体的运动千变万化，很难用一种简单的模型去描述，也很难用一种单一的算法来搜索最佳运动矢量，因此实际上大多采用多种搜索算法相组合的办法，可以在很大程度上提高预测的有效性和鲁棒性，如图 11-6 所示为 H.264 视频标准推荐的运动估计算法，全称是“非对称十字型多层次六边形格点运动搜索算法”。

H.264 的运动估计算法利用了多种搜索方式。

- 非对称十字形搜索，水平搜索多于垂直方向的搜索。
- 正方形区域搜索。
- 中六边形模板、十字模板。

有时根据性能需求简化该图，实现一部分搜索方式。经过实验证明对于如 salesman 等中小运动序列，其速度可接近全局搜索算法的 400 倍，接近三步搜索算法的 4 倍；而对于大运动序列，如 table tennis，该算法则会自动调节搜索点数以适应复杂的运动。当从总体上考察速度方面的性能时，经试验得知，该算法平均速度是全局搜索算法的 287.4 倍，三步搜索的 2.8 倍。

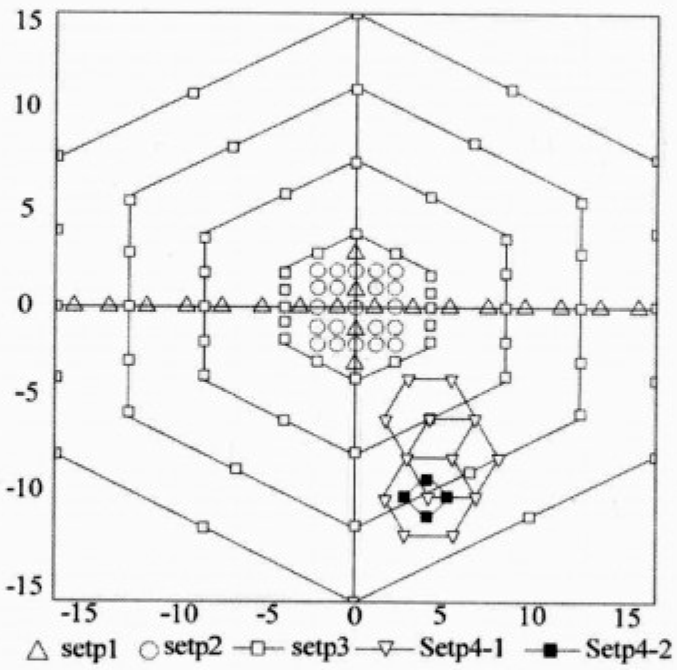


图 11-6　H.264 混合运动估计算法搜索过程图

④ 运动补偿插值

根据匹配准则得到最佳匹配宏块及其运动矢量，作减运算得到残差。在计算残差时，需要根据运动矢量和参考帧类型作插值补偿，即在参考帧中，对应宏块位置偏移运动矢量大小得到匹配宏块，然后根据运动矢量选择插值帧：水平插值、垂直插值、斜角插值，最后做减法运算，得到残差完成运动补偿。

事实上，在进行运动估计时也并不是单一使用上述某一类搜索算法，而是根据各类算法的优点灵活组合采纳。在运动幅度比较大的情况下可以采用自适应的菱形搜索法和六边形搜索法，这样可以大大节省码率而图像质量并未有所下降。在运动图像非常复杂的情况下，采用全局搜索法在比特数相对来说增加不多的情况下使得图像质量得到保证。

2．变换编码与量化编码

1）变换编码

变换编码是把时域、空域的信号通过数据工具转换到另外一个域，在这个域中，信号的能量重新分布，能量更集中，从而便于编码压缩。根据统计和经验得知，平坦区域或内容缓慢变化区域占据一幅图像的绝大部分，而细节区域或内容突变区域则占小部分。也可以说，图像中直流和低频分量占大部分，高频分量占小部分。这样，空间域的图像变换到频域或所谓的变换域，会产生相关性很小的一些变换系数，并可对其进行压缩编码。视频编码中的变换编码的数据源包括两种：一是图像数据本身如帧内编码，一是图像残差。

变换中的正交变换可用于图像编码。自 1968 年利用快速傅立叶变换（FFT）进行图像编码以来，出现了多种正交变换编码方法，如 K-L 变换、离散余弦变换（DCT）、小波变换等。其中，编码性能以 K-L 变换最为理想，但缺乏快速算法，且变换矩阵随图像而异，不同图像需计算不同的变换矩阵，因而只用来参考比较。DCT 编码性能最接近于 K-L 变换，略次而已，具有快速算法，广泛应用于图像编码，如 MPEG-1/2/4、H.261/263 的浮点 DCT 变换和 H.264 中的整数 DCT 变换。JPEG 采用了 DCT 变换，JPEG 2000 采用了小波变换。视频算法中的 DCT 原理如下所示，其中 $f(x,y)$ 为 N×N 数字图像序列，DCT 和 IDCT 变换核如下公式所示。

二维 DCT 变换表示为：

$$F(u,v)=\frac{2}{N}c(u)c(v)\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}f(x,y)\cos\left[\frac{\pi}{2N}(2x+1)u\right]\cos\left[\frac{\pi}{2N}(2y+1)v\right] \quad (11\text{-}4)$$

$$c(u)=\begin{cases}\frac{1}{\sqrt{2}} & u=0\\ 1 & u\neq 0\end{cases},\quad c(v)=\begin{cases}\frac{1}{\sqrt{2}} & v=0\\ 1 & v\neq 0\end{cases},\ \text{其中}\ u,v=0,1,\cdots N-1 \quad (11\text{-}5)$$

二维 IDCT 变换表示为：

$$f(x,y)=\frac{2}{N}\sum_{u=0}^{N-1}\sum_{v=0}^{N-1}c(u)c(v)F(u,v)\cos\left[\frac{\pi}{2N}(2x+1)u\right]\cos\left[\frac{\pi}{2N}(2y+1)v\right] \quad (11\text{-}6)$$

$c(u)$和 $c(v)$同式（11-5），其中 $x,y=0,1,\cdots N-1$。

DCT 和 IDCT 也可写成矩阵形式：

$$F = AfA^T \quad (11\text{-}7)$$

$$f = A^T FA \quad (11\text{-}8)$$

对图像做变换编码，最理想的变换操作应对整个图像进行，以便去除所有像素间的相关性。但这样的操作计算量太大，需要的临时存储区太大。实际上，往往把图像分为若干块如 16×16 的宏块，以宏块为单位进行 DCT 变换，通常为 16×16 或 8×8 的块。但是也正因为以宏块为处理单位，解码后的图像有明显的块效应，这就需要用户在解码时增加后处理以提高图像质量。H.264 视频编码算法就在编码内部开启了环路滤波功能，提高重建参考帧的质量。

传统的 DCT 不是完全可逆的，所以解码出的图像有“拖尾效应”及误差积累。即 I 帧间隔不能太大，否则 DCT 的不可逆操作 IDCT 产生更多的图像误差积累。H.264 提供了整数 DCT/IDCT，其反变换是完全可逆的。H.264 图像效果在同条件下，明显优于其他以往标准算法，很重要的一个特点就是变换采用了整数 DCT。

2）量化编码

量化编码是把 DCT 系数除以一个常量，经过量化后的结果是量化步长的整数倍或为更多的零值，从而达到了压缩的目的。如 H.263 编码方式是 16×16 宏块的 DCT 系数除以同一个量化步长，即均匀量化；而 MPEG 的量化方式是采用量化矩阵的方法，即不同重要性的 DCT 系数除以不同的量化步长。在反量化时，由于量化过程取整或四舍五入，而无法完全恢复原 DCT 系数，从而产生失真。量化是视频编码失真的根本原因。量化的原理公式如式 11-8 所示。

$$q(x,y)=round(\frac{F(x,y)}{Q}+0.5) \quad (11\text{-}8)$$

反量化的原理公式如式 11-9 所示。

$$F'(x,y)=q(x,y)\times Q \quad (11\text{-}9)$$

由于平台没有提供专用的除法指令，所以视频编码标准中量化/反量化的实际应用过程中，采用了查表或移位的方式来实现。如 H.263 的量化方式就是采用的查表加移位的方式。量化步长越小，等级越多，则码流大小等级越丰富。如 XviD 的 MPEG-4 算法的 H.263 量化方式有 32（031）个量化等级，而 H.264 有 52（051）个量化等级。

3）之字形扫描

DCT 变换使空域的图像能量重新分布，主要集中在左上角，右下方大部分是较小值或零值。为了把左上方的重要的非零值按照一维的顺序串联起来，需要把 DCT 的系数按照一定的次序重新放置。“之”字形扫描方式就可以把图像块的这些非零值串接起来，从而把更多的零值丢弃，这样便于后续的熵编码，如图 11-7 所示。

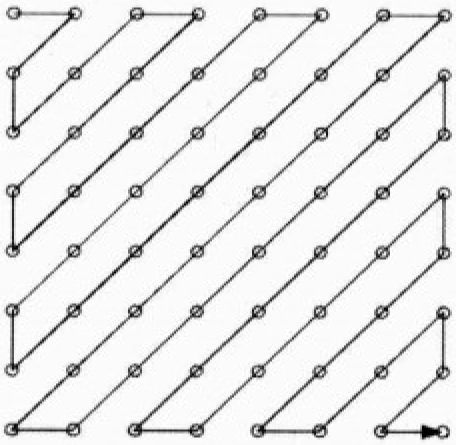


图 11-7 “之”字形扫描

3. 视频统计熵编码

经过变换、量化后，图像数据已经经过了一定的压缩，而图像的 DCT 量化系数已经是相关性较小、系数都很重要及结果概率分布有一定规律。混合视频编码框架最后的一个模块是熵编码，即统计编码：符号出现概率大的用短码字来表示，符号出现概率小的用长码字来表示，从而起到了数据压缩的目的，所以统计编码又称为变长编码 VLC（Varaible Length Coding）。Huffman 编码、算法编码等都是统计的熵编码。实用的视频编码算法中，熵编码采用的是查表方式。MPEG1/2/4 和 H.261/3 等采取的是 Huffman 编码，H.264 中 baseline 级的熵编码是上下文自适应的可变字长编码 CAVLC，即改进的 Huffman 编码，H.264 中的 Main 或 Extended 级的熵编码采取的是上下文自适应的二进制算术编码 CABAC。

熵编码针对图像或残差，而视频编码算法的其他信息如运动向量、直流分量、编码模式等控制信息的编码方式采用其他方法，如 H.264 中采取的是哥伦布编码（Golomb）。

熵编码是可逆编码，所以在视频编码算法框架中，视频数据的失真主要存在于变换、量化。如果变换可逆则只存在于量化。无损视频编码中，就是采用的统计熵编码，如 Huffyuv CODEC 就是无损视频编/解码算法的一个案例。

11.3 编/解码技术标准

音/视频编/解码技术现在已经形成了一个产业，并且已经颁布了多个技术标准。制定音/视频编/解码的国际组织主要有两个：ITU-T 国际电信联盟（以前称为国际电报电话咨询委员

会，(Consultative Committee of International Telegraph and Telephone，CCITT）和 ISO/IEC（国际标准化组织 ISO、国际电工委员会 IEC），他们既有自己独立制定的技术标准又有联合制定的技术标准。编/解码标准涉及的数据有语音、音频、静态图像、数字视频等，且有的标准既涉及图像又涉及声音。

- 语音编/解码标准：PCM、ADPCM、G.711、G.721、G.723、G.726、G.728、G.729 等。
- 音频编/解码标准：MP3、MPEG1/2/4 音频、AVS 音频、WMA、RA、RM 等。
- 静态图像标准：JPEG、GIF、JBIG（二值图像）、JPEG 2000 等。
- 视频编/解码标准：MPEG-1、MPEG-2、MPEG-4、H.261、H.263、H.264、AVS 等。

语音和音频编/解码算法相对于图像、视频算法其运算量和复杂度都要小得多，并且其发展和更新也较慢。视频编/解码标准更新快、应用领域多（有很多应用系统可以没有音频），并且算法复杂，且实现方式有多种：ASIC 专用芯片、ARM 嵌入式处理器、X86 通用处理器（Intel/AMD CPU）、DSP 数字信号处理器、FPGA 现场可编程门阵列。可编程实现视频编/解码是应用的主流方式，如 X86 软件实现、可编程 DSP 数字媒体处理器硬件编/解码。

11.3.1 静态图像编码标准

静态图像的编码压缩主要应用在 PC、DC 数字照相机或网络传输等场合，这些场合的应用对压缩实时性没有太高要求，压缩效率也不是很苛刻，而更加关心解码重建的图像质量。早期的静态图像标准是 JPEG（Joint Photographic Expert Group），现在的静态图像压缩标准更多地采用 JPEG 2000。

1．JPEG

ISO 和 CCITT 于 1986 年年底成立了“联合图像专家组”(Joint Photographic Experts Group，简称 JPEG)，致力于研究静止图像压缩算法的国际标准，JPEG 的标准号为 ISO 10918-1。JPEG 在视觉效果不受到损失的前提下，可以达到 1520 的压缩比。如果图像质量稍微牺牲一点，则可以达到 40 倍或者更高的压缩比。JPEG 压缩算法不是只有一种模式，它可以细分为 4 种模式，分别为基本压缩模式（Baseline Sequential Coding）、渐进式压缩模式（Progressive Coding）、无损编码模式（Lossless Coding）、阶层模式（Hierarchical Coding）。后 3 种压缩模式都是通过基本压缩模式加以改良而来的。

如果待编码的图像是 RGB 格式，则还需要在 JPEG 编码前转换为 YUV 分量形式，JPEG 本身不做这部分工作。JPEG 对 3 个 YUV 分量的处理方式基本相同。

JPEG 的基本压缩模式的编码和解码处理过程如下。

1）JPEG 编码

JPEG 编码器结构框图如图 11-8 所示，原始图像数据以 8×8 数据子块为处理单位，子块首先经前向 DCT（FDCT）变换，图像块能量重新分布，低频集中在左上角。其 DCT 系数均匀量化后（Quantizer)，大部分的系数变为了零，这也正是编码所期望的。鉴于零系数的分布特征及为更有效地压缩，量化后的系数做 Z 型扫描（Zzag)，最后做 Huffman 编码（HuffmanC）存储到一个文件中。循环处理直到把完整的图像压缩完为止。其中量化和 Huffman 熵编码是查表编码，在编码前先创建这两个表。

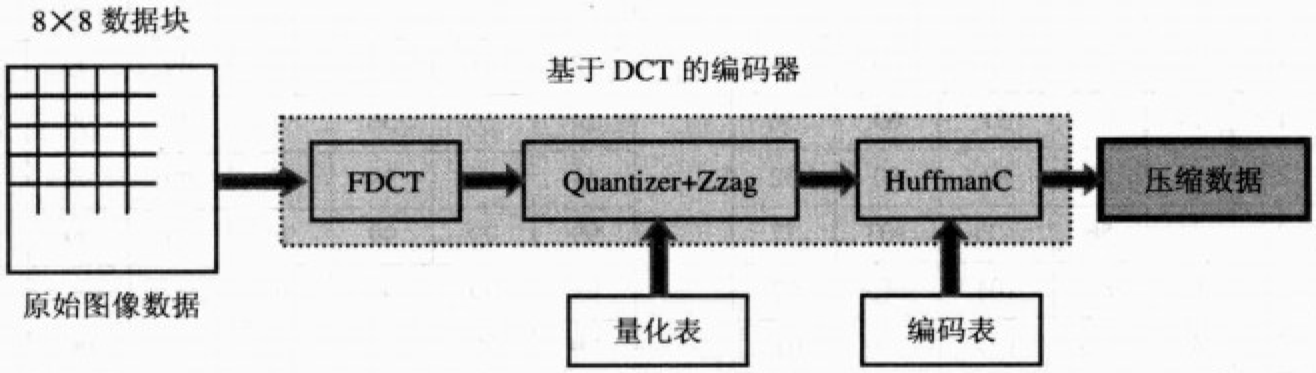


图 11-8　JPEG 编码结构框图

- FDCT：前向离散余弦变换。
- Quantizer+Zzag：量化器及 Z 型扫描。
- HuffmanC：霍夫曼编码器。

2）JPEG 解码

解码器结构功能框图如图 11-9 所示，取得一个压缩的 8×8 的数据块（在压缩 8×8 单元时做标记 EOB 或其他符号以示是处理单元结束），查 Huffman 编码表进行熵解码（Huffman_D）、反 Z 型扫描（I_Zzag）和反量化（I_Quantizer），最后逆离散余弦变换（IDCT）生成重建图像，循环操作直至重建整幅图像。

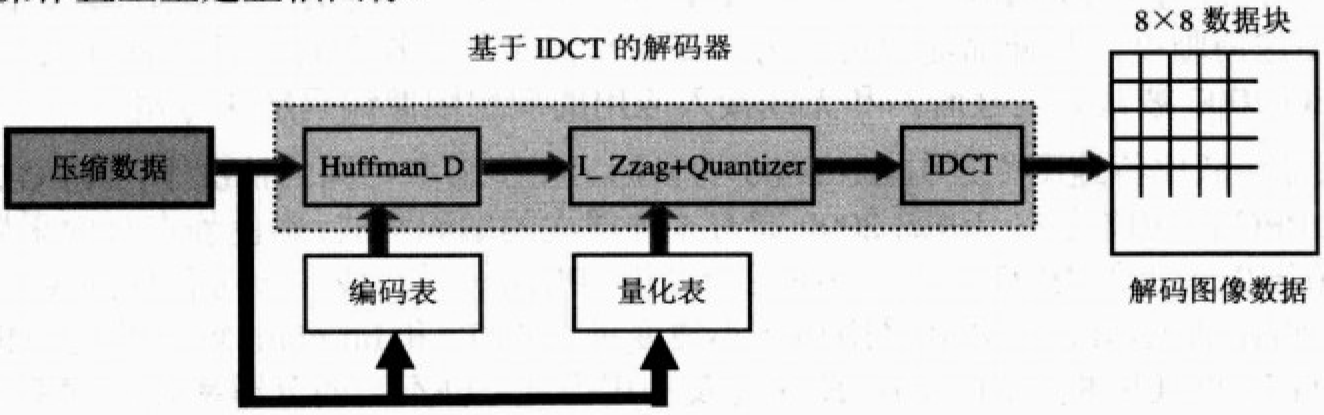


图 11-9　JPEG 解码结构框图

- Huffman_D：霍夫曼熵解码。
- I_ Zzag+Quantizer：反 Z 型扫描和反量化。
- IDCT：逆离散余弦变换。

3）亮度和色度量化表

编码算法中的量化对压缩效率和图像质量起着关键作用，JPEG 推荐了经验亮度和色度的量化表如图 11-10、图 11-11 所示，这里的量化技术是非均匀量化。亮度量化表中总趋势是左上角量化步长短，右下角步长较长，这样对 DCT 系数量化（整除）后，块的右下角绝大部分成了 0 值。色度量化表较规整，绝大部分为 99，仅左上角量化步长较短。

4）JPEG 应用

JPEG 的优良品质使得它在短短的几年内就获得了极大的成功。目前绝大多数的静态图像都是采用的 JPEG 的压缩标准，各类浏览器、图片浏览处理工具均支持 JPEG 图像格式。同时由于 JPEG 格式的文件尺寸较小，Web 页有可能以较短的下载时间提供大量精美的图像，使得 JPEG 成为网络上最受欢迎的图像格式。

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	56	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

图 11-10　亮度量化表

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

图 11-11　色度量化表

自 JPEG 制定后的十多年，其他更有效的图像压缩技术得到快速发展，如小波变换方法、分形压缩等。发展最成熟和应用性能最好的静态图像编码方法是小波变换方法，由此 JPEG 组织于 2000 年制定并颁布了 JPEG 2000，其核心就是小波变换方法。

2．JPEG 2000

随着多媒体应用领域的激增，传统的 JPEG 压缩技术已经无法满足人们对多媒体数据的要求，于是，一种功能更强大、效率更卓越的静止图像压缩标准被提到制定日程上，这就是 JPEG 2000，标准号为 ISO/IEC 15444lITU-T T.800。JPEG 2000 采用改进的压缩技术来提供更高的解码图像清晰度，其伸缩能力可以为一个文件提供从无损到有损的多种画质和解码重建选择。JPEG 2000 被认为是互联网和无线接入应用的理想图像编码解决方案。

“高压缩、低比特速率”是 JPEG 2000 的目标。在压缩率相同的情况下，JPEG 2000 的信噪比将比 JPEG 高 30%左右。JPEG 2000 拥有 5 种层次的编码形式：彩色静态画面采用的 JPEG 编码、二值图像采用的 JBIG、低压缩率图像采用 JPEGLS 等，成为应对各种图像的通用编码方式。在编码算法上，JPEG 2000 采用离散小波变换（DWT）和 bit plain 算术编码（MQ coder）。此外，JPEG 2000 还能根据用户的线路速度及利用方式，以不同的分辨率及压缩率发送图像。JPEG 2000 的文件采用扩展名“.j2k”。

1）JPEG 2000 的原理

JPEG 2000 与传统 JPEG 的最大不同在于它放弃了 JPEG 所采用的以离散余弦转换 DCT 为主的块编码方式，而改用以小波转换（Wavelet Transform）为主的多解析编码方式。小波转换的主要目的是将影像的频率成分抽取出来。

2）JPEG 2000 的优点

- JPEG 2000 作为 JPEG 升级版，高压缩（低比特速率）是其目标，其压缩率比 JPEG 高 30%左右。
- JPEG 2000 同时支持有损和无损压缩，而 JPEG 只能支持有损压缩。无损压缩对保存一些重要图片十分有用。
- JPEG 2000 能实现渐进传输，这是 JPEG 2000 的一个极其重要的特征。也就是我们对 GIF 格式影像常说的“渐现”特性。它先传输图像的轮廓，然后逐步传输数据，不断提高图像质量，让图像由朦胧到清晰显示，而不必是像现在的 JPEG 一样，由上到下慢慢显示。
- JPEG 2000 支持所谓的“感兴趣区域”特性，读者可任意指定影像上感兴趣区域的

压缩质量，还可以选择指定的部分先解压缩。这样我们就可以很方便地突出重点了。

3）JPEG 2000 的应用

JPEG 2000 的应用领域可大致分成两部分，一为传统 JPEG 的市场，如印表机、扫描器和数码相机等；一为新兴应用领域，如网络传输、移动图像通信和医疗影像等。目前对 JPEG 2000 热情最大的当然就是那些数字照相机厂商。JPEG 2000 和 JPEG 相比优势明显，且向下兼容，因此取代传统的 JPEG 格式指日可待。

ASIC 芯片厂商看到了 JPEG 2000 的潜在优势，纷纷推出了编/解码芯片 ADV 202、ADV 212 等，ADI 的这些芯片得到了广泛的工程应用。

4）JPEG 2000 组织结构

JPEG 2000 标准包括 10 个组成部分。

第 1 部分：JPEG 2000 图像编/解码系统，即 JPEG 2000 的核心系统。

第 2 部分：扩展系统，在核心系统基础上提供了一些可选技术。

第 3 部分：在核心系统基础上定义了运动 JPEG 2000（Motion JPEG 2000）。

第 4 部分：定义算法适应性的测试。

第 5 部分：定义高质量免费参考软件。一个是 Java 编写的 JPEG 2000 第 1 部分，称为 JJ2000；另一个是用 C 实现，称为 Jasper。

第 6 部分：定义复合图像文件格式，用于印刷和传真。

第 7 部分：技术报告，介绍实现第 1 部分的最小支持环境。

第 8 部分：安全性（JPSEC）。

第 9 部分：交互性和传输协议。

第 10 部分：三维编码（JP3D）。

5）JPEG 2000 编码模式

JPEG 2000 有两种编码模式。

- 基于 DCT 的编码模式：采用现在的传统 JPEG。
- 基于小波的编码模式：包括有损和无损的小波变换 DWT。

JPEG 2000 基于 DCT 的编码模式是为了兼容 JPEG，但对算法进行了更新或改进。基于小波的编码模式采用了基于离散小波变换（Digital Wavelet Transform，DWT）技术、标量量化、上下文建模、算术编码及后压缩率配置等新技术。

DWT 属于基二算法，既可以用可逆的 Le Gall 分支滤波器实现，也可用不可逆的 Daubechies 双正交分支实现，前者供无损编码，后者提供高压缩率，但不支持无损编码。量化器允许嵌入死区（零周围的值量化为零）标量量化方法，并与其他各子带独立。每一个子带划分成若干矩形块，称为 JPEG 2000 的码块。码块的典型规格是 64×64，用上下文建模（Context Modeling）和位面（Bit Plane）算术编码方法进行熵编码。编码的数据采用后置压缩率配置组织成等级层（Quality Layers），输出到数据包形式的码流（Code Stream）中。产生的码流是可分析的，并能解析、分层、位置或分量累进等。

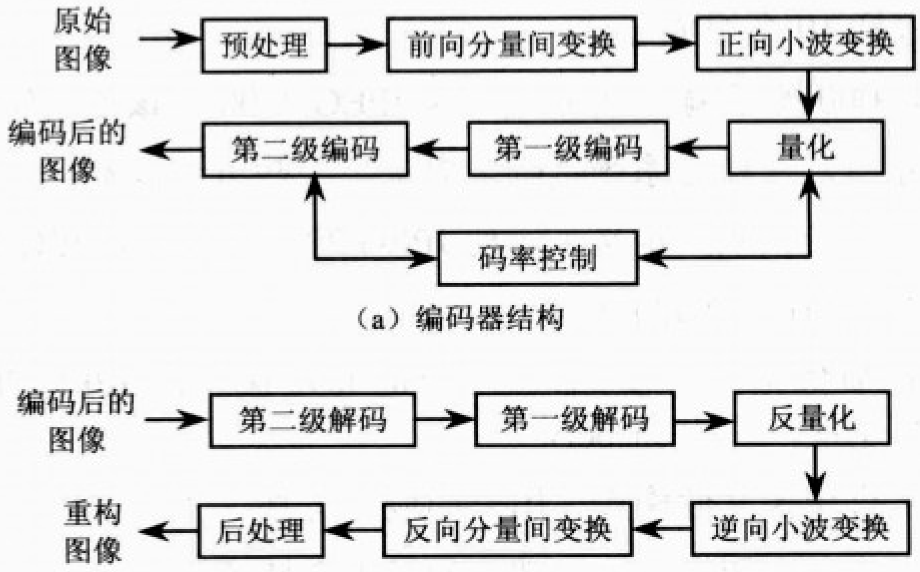
此外，JPEG 2000 还提供很多其他功能。其中很多是算法固有的，例如，因为编码过程

对立于码块和码流的数据打包结构，就可能实现随机存取图像数据。再如，也可能编码感兴趣的任意形状的区域（Regions of Interest，ROI）。由于子带是多个位面编码的，因此使在码流中的其余图像之前选择图像的区域成为可能。通过调节（Scale）该子带的抽样，使首先编码的位面只包含 ROI 信息，随即编码的位面只包含背景信息。解码器唯一需要的是要接收到该子带抽样的调节因子，就能基于样本幅度转化这些样本。

JPEG 2000 支持的其他功能还很多，例如错误恢复、随机存取、多种成分的图像、调色的色彩、被压缩域的无损倒装和抽样交替等。

6）JPEG 2000 核心编/解码器

JPEG 2000 的核心编/解码器的原理框图如图 11-12 所示。



（a）编码器结构

（b）解码器结构

图 11-12　JPEG 2000 编/解码器结构

编码器首先对原始图像进行离散小波变换（DWT），变换后的系数经过量化、熵编码，最后组织成 JPEG 2000 码流（比特流）格式。解码器是编码器的逆过程，即码流首先进行熵解码、反量化，然后进行逆向离散小波变换，最终形成重构图像数据。

下面介绍 JPEG 2000 编/解码的算法处理过程。

预处理前向分量变换前对源图像的所有工作全部称为预处理。这个阶段包括将源图像进行瓦片（tile）分割和直流电平移动。瓦片分割是指将源图像分割成为不重叠的矩形块，这些矩形块可以进行单独压缩编码，就像它们是完全独立的图像。分割的目的是降低压缩过程所需的内存资源。直流电平移动可以使样本的动态范围基本关于零对称，使在进行离散小波变换后的系数的动态范围不会过大，从而有利于编码。

分量变换指对具有多个分量（RGB/YUV）的图像先经过某种变换来降低这几个分量的相关性，达到提高压缩效率的作用。JPEG2000 标准中定义了两种分量间变换：可逆分量变换和不可逆分量变换。不可逆变换的实质就是从 RGB 彩色空间变换到 YCbCr 色彩空间。

不可逆分量变换的公式为：

$$\begin{bmatrix} Y_0(x,y) \\ Y_1(x,y) \\ Y_2(x,y) \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.16875 & -0.33126 & 0.5 \\ 0.5 & 0.41869 & -0.08131 \end{bmatrix} \begin{bmatrix} I_0(x,y) \\ I_1(x,y) \\ I_2(x,y) \end{bmatrix} \tag{11-10}$$

可逆分量变换的公式为：

$$Y_0(x,y)=\frac{1}{4}[I_0(x,y)+2I_1(x,y)+I_2(x,y)]$$

$$Y_1(x,y)=I_2(x,y)-I_1(x,y)] \qquad (11\text{-}11)$$

$$Y_2(x,y)=I_0(x,y)-I_1(x,y)]$$

离散小波变换的目的是去除每个子图像内部像素之间的相关性，尽可能地将信息集中到少的变换系数上。JPEG 2000 采用小波变换编码，与离散余弦变换相比，离散小波变换具有很好的局部性，能针对不同类型图像的不同区域采用不同的空间分辨率，从而有可能得到较高的压缩比，而且可以实现无损压缩的功能。

JPEG 2000 标准中的离散小波变换是对每个瓦片的每个分量进行的，每一个瓦片分量独立地进行变换产生一系列的二维子带信号，每一个子带代表了原图像在不同频带、不同分辨率的表现。对每个瓦片进行 Mallat 塔式小波分解，选用的滤波器是用于浮点运算的 Daubechies(9, 7)滤波器和用于定点运算的 Le Call(5, 3)滤波器。

图 11-13 是正向小波变换的算法流程的核心部分，实际正向离散小波变换就是反复进行该图所示的过程，直至分解到指定的分辨率等级。

图 11-13　JPEG 2000 中的小波变换

小波变换的系数经过量化，瓦片分量数据变换成子带形式，若进行 N 级小波分解，则可以得到 3N+1 个子带，每个子带系数都反映了图像不同频率的特征，JPEG 2000 标准中采用均匀量化，但不同子带使用不同的量化步长。

图像小波系数数据量化后，变换系数矩阵成为一个整数矩阵，接下来要进行的是编码过程，JPEG 2000 编码器采用的是改进的 EBCOT（Embedded Block Coding with Optimized Trunction），它是最优截断的嵌入式块编码。EBCOT 算法借鉴了 LZW、SPIHT 和 LZC 的许多思想，提供了更好的分级和随机访问的功能。

第一级编码，经过量化后的系数要组织成为码块（Code Block）、位平面（Bit Plane）和编码通道（Coding Pass）。这一级实质就是对各种编码要素进行熵编码，主要是基于上下文的二进制算术编码。

经过小波变换和量化，瓦片分量矩阵成为整数系数的子带矩阵。每个子带又要被划分为大小相同的矩形码块，码块的宽和高都要是 2 的幂，典型值是 32×32 或 64×64，每个码块将被独立编码。每个码块又可以分成位平面，即比特层。从最高有效位平面开始逐平面编码直到最低位平面。

扫描过程：JPEG 2000 中每个位平面上的比特将被分在 3 个不同的编码通道上，每个比特只在其中一个编码通道上编码，最终形成 3 个编码通道。这 3 个通道依次为重要性传播通道（Significance Propagation Pass）、幅度细化通道（Magnitude Refinement Pass）、清除通道（Cleanup Pass）。编码器按顺序扫描一层位平面上所有比特 3 遍，每 4 个系数为 1 列，从第一个系数开始。先扫位平面的 1～4 行，然后是 5～8 行，如图 11-14 所示。

0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60
1	5	9	13	17	21	25	29	33	37	41	45	49	53	57	61
2	6	10	14	18	22	26	30	34	38	42	46	50	54	58	62
3	7	11	15	19	23	27	31	35	39	43	47	51	55	59	63
64	…														
65	…														

图 11-14　小波系数扫描模式

第一遍扫描形成重要性传播编码通道；第二遍扫描形成幅度细化编码通道；第三遍扫描形成清除编码通道。位平面上的比特在哪个通道上编码取决于编码时该比特所在的系数有效性状态，以及它周围的 8 个相邻比特所在系数的有效性状态。码块中每个系数都对应了 1 个比特表示该系数的有效性状态，有效性状态初始时为 0（称为无效），随着编码的进行，当该系数的第一个非零的比特也就是最高位被编码后，其有效性状态变为 1（称为有效）。

第二级编码过程实际上是分层（Layer）、打包（Packet）从而形成码流（Codestream）的过程。这层编码最终完成了 JPEG 2000 标准具有的许多优良性能，如压缩质量和分辨率可分级性。

JPEG 2000 的编码技术如上所述，其中技术核心是离散小波变换和 EBCOT 编码。离散小波变换是 JPEG 2000 码流具有分辨率可分级的基础，而 EBCOT 编码是 JPEG 2000 实现信噪比可分级的基础。

尽管 ADI 的 JPEG 2000 编/解码芯片 ADV 202/212 既支持静态图像编码又支持视频编码，但因为都是帧内编码，所以编码效率很低。所以针对视频有专门的压缩算法。如 ISO 的 MPEG 系列和 ITU 的 H.26X 系列。

11.3.2　MPEG-4/H.264视频算法

在视频编/解码国际标准中，MPEG-4 和 H.264 是当今视频应用和开发的主流。MPEG-4 是 ISO 组织制定的音频编/解码算法，主要针对网络、视频会议和可视电话等低码率传输应用。H.264 是 ITU-T 和 ISO 联合组织 JVT（Joint Video Team）制定的视频编/解码算法，在 MPEG-4 中是 Part-10，称为 AVC，在 ITU 中称为 H.264。所以 H.264/AVC 是同一个标准在不同组织中的不同称谓。主要的视频算法发展历程如图 11-15 所示。

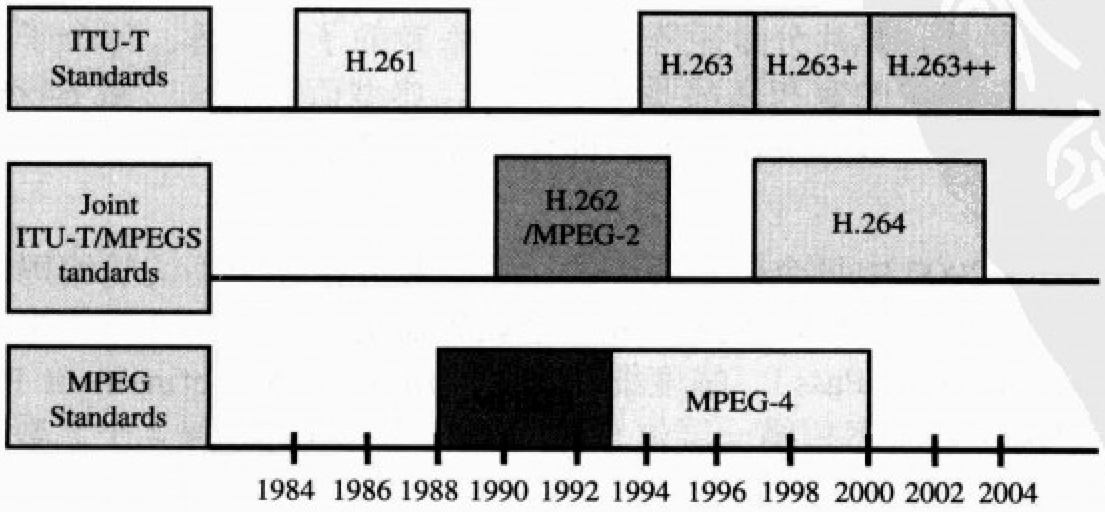


图 11-15　视频算法发展历程

下面分别介绍这两种视频算法的来源、应用和主要特点。后续章节分解以具体的开发案例讲解算法流程、开发应用过程和技术要点等。

1. MPEG-4 视频算法

MPEG-4 是由 MPEG（Moving Picture Expert Group，动态图像专家组）制订的一个 ISO/IEC 标准，该委员会曾经制订获得艾美奖的标准——MPEG-1（VCD）和 MPEG-2（DVD）。这些标准使在 CD-ROM 和数字电视上制作交互视频成为可能。MPEG-4 是国际组织和遍布世界的几百位研究者与工程师合作的成果。MPEG-4 的标准号是 ISO/IEC 14496，定稿于 1998 年 10 月，于 1999 年 1 月成为国际标准。完全向后兼容的 MPEG-4 第 2 版于 1999 年底定稿，于 2000 年初被正式接受为国际标准，有些细节仍在制订过程中。MPEG-4 算法标准适应范围宽，档级多，从兼容传统的视频编码到高级的对象编码。视频对象编码是把视频场景按照对象目标进行分割，一般包括场景和场景的各个目标。对不同的目标对象选用不同的编码技术，从而达到系统的高效的视频编码。但是由于目前视频对象分割仍然是个难点，还处在研究阶段或已有技术尚未成熟，MPEG-4 的应用和开发仍然是基于像素的传统视频编码。只是把整个帧看做一个对象 VOP，分别编码纹理信息（残差或图像本身）和运动信息（运动向量）。

MPEG-4 提供了大量丰富的音频视频对象的编码工具，能够满足多样应用需求。而对于某一特定的应用，只有一部分编码工具被采用，框架（Profile）就是针对特定的应用确定要采用的编码工具，它是 MPEG-4 提供的工具集的一个子集。每一个框架又有一个或多个级别（Level）来限制算法的复杂度。MPEG-4 共有 4 类框架：视频框架、音频框架、图形框架和场景描述框架，其中视频框架又包含 5 个级别。

- Simple Visual Profile：提供矩形视频对象高效有容错能力的编码功能，适合应用于移动网络。
- Simple Scalable Visual Profile：在 Simple Profile 基础上增加了对象时域和空域扩展编码功能，应用于提供多级服务质量的应用，如 Internet 和软件解码。
- Core Visual Profile：在 Simple Profile 基础上增加了任意形状对象编码和时域扩展编码功能，适用于相对简单的内容交互应用，如 Internet 多媒体应用。
- Main Visual Profile：在 Core Profile 基础上增加了 Sprite 对象编码功能，适用于交互式应用和娱乐级别质量的视频广播和 DVD 应用等。
- N-Bit Visual Profile：在 Core Profile 基础上增加了具有不同像素深度（4～12bits）的视频对象编码功能，适用于监控应用。

MPEG-4 之前的视频标准：MPEG-1/2、H.261、H.263 都是采用第一代压缩编码技术，基于图像信号的统计特性来设计编码器，属于波形编码的范畴。第一代压缩编码方案把视频序列按时间先后分为一系列帧，每一帧图像又分成宏块以进行预测、运动补偿和编码，这种编码方案存在以下缺陷。

- 将图像分成相同大小的块，在高压缩比情况下会出现严重的块效应。
- 无法对视频内容进行访问、编辑和回放等操作。
- 未充分利用人类视觉系统（Human Visual System，HVS）的特性。

MPEG-4 则代表了基于模型/对象的第二代压缩编码技术，它充分利用了人眼视觉特性，抓住了图像信息传输的本质，从轮廓、纹理思路出发，支持基于视觉内容的交互功能，这适

应了多媒体信息的应用由播放型转向基于内容的访问、检索及操作的发展趋势。

AV 对象（Audio Visual Object，AVO）是 MPEG-4 为支持基于内容编码而提出的重要概念。对象是指在一个场景中能够访问和操纵的实体，对象的划分可以其独特的纹理、运动、形状、模型和高层语义为依据。在 MPEG-4 中所见的音/视频已不再是过去 MPEG-1/2 中图像帧的概念，而是一个个视听场景（AV 场景），这些不同的 AV 场景由不同的 AV 对象组成。AV 对象是听觉、视觉、或者视听内容的表示单元，其基本单位是原始 AV 对象，它可以是自然或合成的声音、图像。原始 AV 对象具有高效编码、高效存储与传输及可交互操作的特性，它又可进一步组成复合 AV 对象。因此 MPEG-4 标准的基本内容就是对 AV 对象进行高效编码、组织、存储与传输。AV 对象的提出使多媒体通信具有高度交互及高效编码的能力，AV 对象编码是 MPEG-4 的核心编码技术。

MPEG-4 不仅可提供高压缩率，同时也可实现更好的多媒体内容互动性及全方位的存取性，它采用开放的编码系统，可随时加入新的编码算法模块，同时也可根据不同应用需求现场配置解码器，以支持多种多媒体应用。MPEG-4 采用了新一代视频编码技术，它在视频编码发展史上第一次把编码对象从图像帧拓展到具有实际意义的任意形状视频对象，从而实现了从基于像素的传统编码向基于对象的现代编码的转变。

MPEG-4 除采用第一代视频编码的核心技术如 DCT 变换、运动估计与运动补偿、量化和熵编码外，还提出了一些新的有创见性的关键技术，并在第一代视频编码技术的基础上进行了卓有成效的完善和改进，下面重点介绍其中的一些关键技术。

1）视频对象编码

MPEG-4 实现基于内容操作的前提是把视频/图像分割成不同对象或者把运动对象从背景中分离出来，然后针对不同的对象采用相应的编码方法，以实现高效压缩。因此视频对象提取即视频对象分割，是 MPEG-4 视频编码的关键技术，也是新一代视频编码的研究热点和难点。

视频对象分割涉及对视频内容的分析和理解，这与多种学科如人工智能、图像理解、模式识别和神经网络等有密切的联系。人工智能的发展还不够完善，计算机还不具有观察、识别、理解图像的能力；同时关于计算机视觉的研究也表明要实现正确的图像分割需要在更高的层次上对视频内容进行理解。因此，尽管 MPEG-4 框架已经制定，但至今仍没有通用的有效方法去根本解决视频对象分割的问题，视频对象分割已被公认为是一个具有挑战性的难题，基于语义的对象分割则更加困难。

2）VOP 视频编码

视频对象平面（Video Object Plane，VOP）是视频对象（VO）在某一时刻的采样，VOP 是 MPEG-4 视频编码的核心概念。MPEG-4 在编码过程中针对不同 VO 采用不同的编码策略，即对前景 VO 的压缩编码尽可能保留细节和平滑；对背景 VO 则采用高压缩率的编码策略，甚至不予传输而在解码端由其他背景拼接而成。这种基于对象的视频编码不仅克服了第一代视频编码中高压缩率编码所产生的方块效应，而且使用户可与场景交互，从而既提高了压缩比，又实现了基于内容的交互，为视频编码提供了广阔的发展空间。

MPEG-4 支持任意形状的图像与视频的编/解码。对于任意形状的视频对象。对于极低比特率实时应用如可视电话、会议电视，MPEG-4 则采用 VLBV（Very Low Bit-rate Video，极低比特率视频）核进行编码。

3）视频编码可分级性

随着因特网业务的增长，在速率起伏很大的 IP（Internet Protocol）网络及具有不同传输特性的异构网络上进行视频传输的要求和应用越来越多。在这种背景下，视频分级编码的重要性日益突出，其应用非常广泛，且具有很高的理论研究及实际应用价值，因此受到人们的极大关注。视频编码的可分级性（Scalability）是指码率的可调整性，即视频数据只压缩一次，却能以多个帧率、空间分辨率或视频质量进行解码，从而可支持多种类型的用户的各种不同的应用要求。

MPEG-4 通过视频对象层（Video Object Layer，VOL）数据结构来实现分级编码。MPEG-4 提供了两种基本分级工具，即时域分级（Temporal Scalability）和空域分级（Spatial Scalability），此外还支持时域和空域的混合分级。每一种分级编码都至少有两层 VOL，低层称为基本层，高层称为增强层。基本层提供了视频序列的基本信息，增强层提供了视频序列更高的分辨率和细节。

在随后补充的视频应用框架中，MPEG-4 提出了 FGS（Fine Granularity Scalable，精细可伸缩性）视频编码算法及 PFGS（Progressive Fine Granularity Scalable，渐进精细可伸缩性）视频编码算法。FGS 编码实现简单，可在编码速率、显示分辨率、内容、解码复杂度等方面提供灵活的自适应和可扩展性，且具有很强的带宽自适应能力和抗误码性能。但还存在编码效率低于非可扩展编码及接收端视频质量非最优两个不足。PFGS 则是为改善 FGS 编码效率而提出的视频编码算法，其基本思想是在增强层图像编码时使用前一帧重建的某个增强层图像为参考进行运动补偿，以使运动补偿更加有效，从而提高编码效率。

4）运动估计与运动补偿

MPEG-4 采用 I-VOP、P-VOP、B-VOP 这 3 种帧格式来表征不同的视频编码类型。它采用了 1/2、1/4 像素搜索和重叠运动补偿（Overlapped Motion Compensation）技术，同时又引入重复填充（Repetitive Padding）技术和修改的块（多边形）匹配技术以支持任意形状的 VOP 区域。

同时，为提高运动估计算法精度、降低算法运算量，MPEG-4 采用了 MVFAST（Motion Vector Field Adaptive Search Technique）和改进的 PMVFAST（Predictive MVFAST）方法用于运动估计。对于全局运动估计，则采用了基于特征的快速稳健的 FFRGMET（Feature-based Fast and Robust Global Motion Estimation Technique）方法。

在 MPEG-4 视频编码算法中，运动估计模块最耗时，对编码的实时性影响最大。因此实际应用开发特强调快速算法。运动估计方法主要有像素递归和块匹配法两大类，前者复杂度很高，在实际中应用较少，后者则在 H.263 和 MPEG 中广泛采用。在块匹配法中，重点研究块匹配准则及搜索方法。目前有 3 种常用的匹配准则，常见的运动估计匹配准则有 3 种：MAD、MSE 和 NCCF。由于 MAD 没有乘除操作，不需做乘法运算，实现简单方便，所以使用较多，通常使用求和绝对误差（SAD）代替 MAD。

在选取匹配准则后就应进行寻找最优匹配点的搜索工作。最简单、最可靠的方法是全搜索法（Full Search，FS），但计算量太大，不便于实时实现。因此快速搜索法应运而生，主要有三步搜索法、二维对数法和钻石搜索法，其中钻石搜索法被 MPEG-4 校验模型（Verification Model，VM）所采纳。钻石搜索（Diamond Search，DS）法以搜索模板形状而得名，具有简单、鲁棒、高效的特点，是现有性能最优的快速搜索算法之一。其基本思想是利用搜索模板的形状和大小对运动估计算法速度及精度产生重要影响的特性。在搜索最优匹配点时，选择小的搜索模板可能会陷入局部最优，选择大的搜索模板则可能无法找到最优点。因此 DS 算

法针对视频图像中运动矢量的基本规律，选用了两种形状大小的搜索模板：大钻石搜索模板（Large Diamond Search Pattern，LDSP）和小钻石搜索模板（Small Diamond Search Pattern，SDSP）。

DS 算法搜索过程：开始阶段先重复使用大钻石搜索模板，直到最佳匹配块落在大钻石中心。由于 LDSP 步长大，因而搜索范围广，可实现粗定位，使搜索不会陷于局部最小，当粗定位结束后，可认为最优点就在 LDSP 周围点所围菱形区域中。然后再使用小钻石搜索模板来实现最佳匹配块的准确定位，以不产生较大起伏，从而提高运动估计精度。

5）MPEG-4 开源算法

MPEG-4 算法提供了参考软件 VM 校验模型，但是运行效率很低，无法实用化。MPEG-4 的最早实用化是微软公司开发的 MPEG-4 V1。后来陆续出现了 DivX、OpenDivx、XviD 等优秀的开源算法。微软力推的 WMV9 是 MPEG-4 算法实现。由于历史原因 DivX5.0 就开始不提供开源代码而被公司私有商用化了。现在应用开发、DSP 移植多数是基于 XviD 的。后续章节将专门以 XviD 为案例剖析 MPEG-4 视频编/解码算法。

2. H.264 视频算法

H.264 是 ITU-T 的 VCEG（视频编码专家组）和 ISO/IEC 的 MPEG（活动图像编码专家组）的联合视频组（Joint Video Team，JVT）开发的一个新的数字视频编码标准，它既是 ITU-T 的 H.264，又是 ISO/IEC 的 MPEG-4 的第十部分。1998 年 1 月份开始征集 H.264 算法草案，1999 年 9 月完成第一个草案，2001 年 5 月制定了其测试模式 TML-8，2002 年 6 月 JVT 第 5 次会议通过了 H.264 的 FCD 版，2003 年 5 月，ITU 正式核准颁布。

H.264 算法还是基于块的混合编码技术，编码过程基本同以前的编码标准，只是每个处理模块都进行了技术更新，帧内预测、帧间预测、整数 DCT 变换、环路滤波、熵编码等模块都做了技术提升。

1）NAL 与 VCL

网络适配层 NAL（Network Abstraction Layer）是 H.264 为适应网络传输应用而制定的一层数据打包操作。传统的视频编码算法编完的视频码流在任何应用领域下（无论用于存储还是传输）都是统一的码流模式，视频码流仅有视频编码层 VCL（Video Coding Layer）。而 H.264 根据不同应用增加不同的 NAL 片头，以适应不同的网络应用环境，减少码流的传输差错。

2）帧内预测

为能进一步利用图像的空间相关性，H.264 引入了多模式的帧内预测以提高压缩效率。简单地说，帧内预测编码就是用周围邻近的像素值来预测当前的像素值，然后对预测误差进行编码。预测是基于块的，亮度分量（Luma）块的大小可以在 16×16 和 4×4 之间选择，16×16 块有 4 种预测模式，4×4 块有 9 种预测模式；色度分量（Chroma）预测是对整个 8×8 块进行的，预测模式同亮度 16×16 的 4 种预测模式。

3）帧间预测

帧间预测即传统的运动估计 ME 加运动补偿 MC，H.264 的运动估计更精准、快速，效果更好。

① 多变的宏块大小

传统的运动估计块大小是 16×16，由于运动物体复杂多变，仅使用一种模式效果不好。

H.264 采用了 7 种方式对一个宏块进行分割：16×16、16×8、8×16、8×8、8×4、4×8、4×4，每种方式下块的大小和形状都不相同，这就使编码器可以根据图像的内容选择最好的预测模式。实验表明，与仅使用 16×16 块进行预测相比，使用不同大小和形状的块可以使码率节省 15%以上。

② 更精细的像素精度

H.264 算法中的 Luma 分量的运动矢量 MV 使用 1/4 像素精度。Chroma 分量的 MV 由 Luma MV 导出，由于 Chroma 分辨率是 Luma 的一半（YUV 4:2:0），所以其 MV 精度将为 1/8。这样，精细的预测精度较之整数精度可以使码率节省 20%以上。

③ 更多参考帧

H.264 支持多参考帧预测（Multiple Reference Frames），即可以将多于一个（最多 5 个）的在当前帧之前解码重建的帧作为参考帧产生对当前帧的预测（Motion-compensated Prediction），这特别适用于视频序列中含有周期性运动的情况。

④ 环路滤波

环路滤波（Loop Filter）的作用是消除经反量化和反变换后重建图像中由于预测误差产生的块效应，从而一方面改善图像的主观质量，另一方面减少预测误差。与以往的 Deblocking Filter 不同的是，经过滤波后的图像将根据需要放在缓存中用于帧间预测，而不是仅仅在输出重建图像时用来改善主观质量，也就是说该滤波器位于解码环中而非解码环的输出外，因而得名 Loop Filter。

4）整数变换

传统的 DCT 由于是浮点算法定点实现，所以 IDCT 不是可逆的，由此造成解码图像的周围“拖尾”现象。H.264 对帧内或帧间预测的残差（Residual）进行整数 DCT 变换编码。新标准对 DCT 的定义做了修改，使得变换仅用整数加减法和移位操作即可实现，这样在不考虑量化影响的情况下，解码端的输出可以准确地恢复编码端的输入。此外，该变换是针对 4×4 块进行的，这也有助于减少块效应。为了进一步利用图像的空间相关性，在对色度（Chroma）的预测残差和 16×16 帧内预测的预测残差进行上述整数 DCT 变换之后，标准还将每个 4×4 变换系数块中的 DC 系数组成 2×2 或 4×4 大小的块，进一步做哈达玛（Hadamard）变换。

5）熵编码

对于预测残差，H.264 有两种熵编码的方式：基于上下文的自适应变长码（Context-based Adaptive Variable Length Coding，CAVLC）和基于上下文的自适应二进制算术编码（Context-based Adaptive Binary Arithmetic Coding，CABAC）；如果不是预测残差，H.264 采用 Exp-Golomb 码或 CABAC 编码，这些视编码器的设置而定。

① CAVLC

可变字长编码 VLC 的基本思想就是对出现频率大的符号使用较短的码字，而对出现频率小的符号采用较长的码字。这样可以使得平均码长最小。在 CAVLC 中，H.264 采用若干 VLC 码表，不同的码表对应不同的概率模型。编码器能够根据上下文，如周围块的非零系数或系数的绝对值大小，在这些码表中自动地选择，最大可能地与当前数据的概率模型匹配，从而实现上下文自适应的功能。

② CABAC

算术编码是一种高效的熵编码方案，其每个符号所对应的码长被认为是分数。由于对每

一个符号的编码都与以前编码的结果有关，所以它考虑的是信源符号序列整体的概率特性，而不是单个符号的概率特性，因而它能够更大程度地逼近信源的极限熵，极大地降低码率。

6）开源的 H.264 算法

ITU 提供了 H.264 的参考软件 JM，包括视频编码和解码，工程采用 VC++平台。该工程只能用来验证 H.264 算法及其码流的规范性，不能实用化。开源的 H.264 算法有 T264 视频编码和解码、X264 视频编码、ffmpeg 的 H.264 视频解码。现在大多数的 H.264 应用是基于 X264 移植的，并且该开源算法一直有更新。

11.3.3 AVS国产视频编码标准

AVS 是我国具备自主知识产权的第二代信源编码标准。顾名思义，“信源”是信息的“源头”，信源编码技术解决的重点问题是数字音/视频海量数据（即初始数据、信源）的编码压缩问题，故也称数字音/视频编/解码技术。显而易见，它是其后数字信息传输、存储、播放等环节的前提，因此是数字音/视频产业的共性基础标准。

AVS 具备以下 3 大特点。

- 我国牵头制定的、技术先进的第二代信源编码标准——先进。
- 领导国际潮流的专利池管理方案，完备的标准工作组法律文件——自主。
- 制定过程开放、国际化——开放。

AVS 标准包括系统、视频、音频、数字版权管理等 4 个主要标准和一致性测试支撑的标准。主要应用于数字电视、激光视盘、网络流媒体、无线流媒体、数字音频广播、视频监控等领域。

AVS 与 MPEG 标准一样都采用了混合编码框架，包括变换、量化、熵编码、帧内预测、帧间预测、环路滤波等技术模块。AVS 在此基础上还提出了一批具体的优化技术，在较低的复杂度下实现了与国际标准相当的技术性能，但并未使用国际标准背后的大量复杂的专利。AVS-视频当中具有特征性的核心技术包括 8×8 整数变换、量化、帧内预测、1/4 精度像素插值、特殊的帧间预测运动补偿、二维熵编码、去块效应环内滤波等。AVS 视频编码器框图如图 11-16 所示。

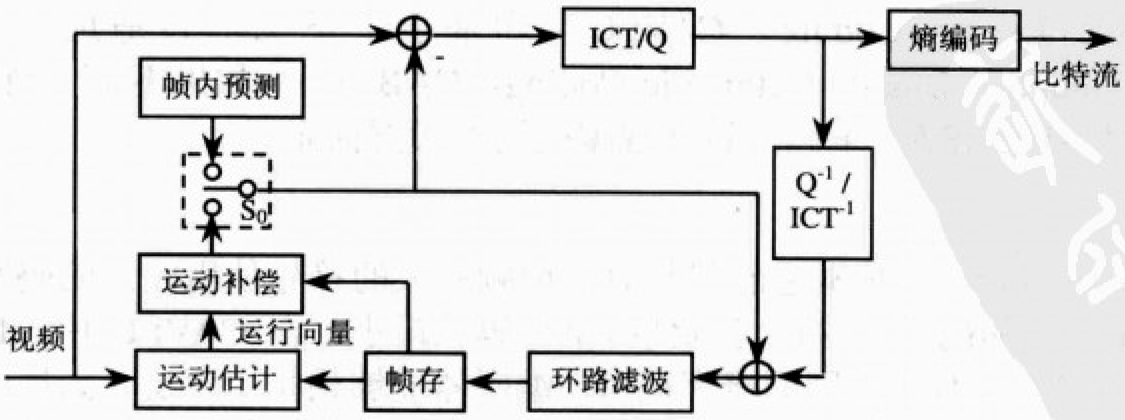


图 11-16　AVS 编码器框图

AVS 的产业化步伐在标准制订过程中已经开始，目前正处在大规模产业化的启动期。

AVS 产业化的主要产品形态包括以下 3 种。

- 芯片：高清晰度/标准清晰度 AVS 编/解码芯片，国内需求量在未来十多年的时间内年均将达 4000 多万片。
- 软件：AVS 节目制作与管理系统，Linux 和 Window 平台上基于 AVS 标准的流媒体回放、编辑等操作软件。
- 整机：AVS 机顶盒、AVS 硬盘播出服务器、AVS 编码器、AVS 高清晰度激光视盘机、AVS 高清晰度数字电视机顶盒和接收机、AVS 手机、AVS 便携式数码产品等。

11.4 编/解码产业及发展

多媒体数据的编码和解码应用已随着网络、计算机和芯片技术的发展，编/解码效率逐渐提高，应用领域也极其广泛。并且已经成为了一个独立的产业，特别是视频编/解码算法的开发和应用。专门开发和提供算法库、根据要求定制算法的软件开发公司也看到了该产业的投资少、效益可观的特点。另外芯片制造商也被该产业的极大诱惑所吸引：应用广泛、需求量大。此外，关于多媒体数据压缩的网络资源也日渐丰富，短小、功能齐全的 DEMO 程序很容易搜集到，开源算法也逐渐增多。

11.4.1 编/解码资源一览

由于编/解码逐渐形成了规范的统一标准，制定组织一般提供了有关算法的参考软件、校验模型，但一般都不实用。而那些专业兴趣爱好者根据编/解码的应用情况，多数有快速的算法实现。常用开源音/视频编/解码 CODEC 如表 11-5 所示。

表 11-5 常用开源音/视频编/解码 CODEC

名称	描述	网址
G.72X 系列	ITU-T 提供参考软件	http://www.itu-t.org
MP3 解码	Libmad	http://www.underbit.com/products/mad/
ffmpeg	多媒体解码 CODEC 集合	http://ffmpeg.mplayerhq.hu/
ffdshow	多媒体解码 CODEC 集合	http://sourceforge.net/projects/ffdshow/
XviD	MPEG-4 编/解码	http://www.xvid.org
DivX	MPEG-4 编/解码	http://www.divx.com
T264	H.264 编/解码	http://sourceforge.net/projects/t264
JM	H.264 编/解码参考软件	http://iphome.hhi.de/suehring/tml
X264	H.264 编码	http://www.videolan.org/developers/x264.html

MPEG-1 和 MPEG-2 编/解码算法源码从 MPEG 主页 http://www.mpeg.org 上可直接获取。

11.4.2 编/解码发展前景

多媒体的编/解码算法从最初的 MPEG-1/MPEG2、H.261/H.263 视频标准到现在的 MPEG-4/H.264、AVS，公司更为关注 MPEG4/H.264 算法的开发应用，但是由于技术专利池的原因，开发商使用这些算法开发相关产品或用户消费该产品都需要支付一定的专利使用费，国产 AVS 专利使用费最低。

1）专利使用费

开发商每销售一台 DVD（MPEG-2 标准）设备，就需要支付大约 2.5 美元的专利费。MPEG-4、H.264：与过去相比，MPEG-LA 虽然降低了解码器的专利费，但开始增收编码器的专利费。如果视频节目运营商提供采用 MPEG-4（已经收费）或者 H.264/MPEG-4 AVC（2006 年 1 月 1 日开始收费）标准的视频节目，则还需要支付加盟费（Participation Fee），即 MPEG 专利技术管理公司 MPEG-LA（MPEG Licensing Administrator）将根据节目、订户和本地发射台数等参数来向运营商收费，每个运营商每年 350 万美元封顶。除此之外，终端用户也需要向 MPEG-LA 交纳每个节目 0.04 美元的费用，该费用将由运营商代收，一并计入收费成本。

国产 AVS 的规则是，要想成为这个标准众多专利权人中的一员，必须在一定程度上牺牲对自身的专利利益，全部服从 AVS 专利管理委员会的管理。专利管理委员会将由政府代表、企业代表和专利权人组成。专利池管理委员会汇集各方的利益，使彼此利益能够得到均衡妥善的处理。据悉，中国政府为了大力推广 AVS，只象征性地收取每台设备 1 元人民币的专利使用费。

2）应用领域

H.264 标准是代表视频编码技术的一项国际标准，于 2003 年 5 月颁布，经过几年的发展，其技术和应用基础都已成熟，在世界范围内有着比较广泛的应用，甚至已成为某些行业的主流标准。目前国内的大部分视频监控、网络流媒体等的产品仍然青睐于这一标准，由于其应用广泛且技术成熟，相应的投入风险也比较低，但是形成规模则需要支付昂贵的专利使用费。而 AVS 的商用之路才刚刚起步，产业链尚在形成之初，因此，AVS 要在普及应用方面追赶上 H.264，还有很长的路要走。但是中国政府力推 AVS，一方面是因为 AVS 完全掌握知识产权，另一方面，其在编/解码效果上与国际标准相差不大。

3）技术对比

从技术层面来讲，AVS 与 H.264 相当，甚至在编码效率和复杂度方面还要胜出。据技术估算，AVS 解码复杂度相当于 H.264 的 30%，AVS 编码复杂度相当于 H.264 的 70%。在国内对 AVS 标准的主客观测试中，AVS 技术均表现出显著的优越性与稳定性，但目前 AVS 只在国内的网络环境下进行过测试，设备商可对网络进行针对性的优化，而国外网络本身要满足多方面的需求，并没有针对国内网络状况进行特定优化。另外，由于国内几大运营商即广电、电信出于某些因素的考虑，还未确定是否选用 AVS。因此，AVS 是否已经超越了 H.264，还需要经过更多样化的测试评估和应用检验，才能更具有说服力。

笔者则认为，国产的 AVS 需要政府强制推行，摆脱受制于人的技术依赖，支持我国的数字音/视频产业，推广 AVS 到全球并纳入国际标准。

11.5 本章小结

本章首先介绍了数字音/视频的一些基础知识，接着介绍了数字化的音/视频在传输、存储前必须做编码压缩，以节省带宽和存储空间。图像、语音、视频等多媒体数据的固有特性：空间和时间上的强相关性，能够在不影响主观应用的情况下，成倍地压缩数据容量。数字多媒体（音频、视频、图像等）的编码算法流程主要采用了混合处理算法：预测、量化和编码。国际组织 ITU-T 和 ISO 独立或联合制定了算法标准：语音算法 G.72X 系列、视频编码 H.26X 系列和 MPEG-X 系列。不同的算法标准应用于不同的领域，MPEG-2 应用于数字电视、DVD 等，MPEG-4 以交互式多媒体应用于网络流媒体、视频监控和视频会议等，H.264 以低码流、高图像质量等特点更适合低码率存储、传输应用。国产的 AVS 与 H.264 编码效果相当，最大特点是专利使用费远低于 MPEG-4/H.264 等国际标准。MPEG-4、H.264 和 AVS 目前的工程应用仍然是基于预测、变换和熵编码的混合编码技术框架，后续的章节将会详细介绍 MPEG-4/H.264 的开发应用。

PDG

第12章

使用 MMX/SSE 2 进行多媒体编程

众所周知，图形处理显示、音/视频编码压缩和解码、音乐合成、图像处理、游戏、语音识别等软件处理算法，一方面数据处理源海量，另一方面应用处理要求实时。这就要求处理器 CPU 有更高的处理性能。Intel 的 CPU 已经被普遍应用到多媒体计算机、图形处理工作站、广告电视的后期编辑处理中。为了提高处理多媒体数据的效率，Intel 先后推出了其 CPU 支持的多媒体扩展汇编指令：MMX（Multi Media eXtension，多媒体扩展指令）、SSE（Streaming SIMD Extensions，单指令多数据流扩展）、SSE 2、SSE 3、SSE 4 等。另一大 CPU 制造商 AMD 也推出了其 CPU 支持的多媒体汇编指令：3DNow!、AMD64 等。本章主要介绍 Intel 的 MMX/SSE 2 的汇编指令和使用这些指令对图像、音/视频进行开发。

本章重点

- MMX/SSE 2 技术基础
- MMX/SSE 2 指令剖析
- 使用 MMX/SSE 2 进行音/视频开发
- MMX/SSE 2 开发常见问题

12.1 MMX/SSE 2技术基础

目前个人计算机（PC）的 CPU 主要有两大厂商 Intel 和 AMD，它们都有自己的多媒体汇编指令，同时也有相同的指令。最早提供 MMX 汇编的是 Intel 的 CPU。下面列出了 Intel 和 AMD 的 CPU 各自支持的多媒体汇编指令。

1）MMX

MMX（Multi Media eXtension，多媒体扩展指令）指令集是 Intel 公司于 1996 年为其 Pentium 系列处理器开发的一项多媒体处理指令增强技术。MMX 指令集中包括了 57 条多媒体指令，通过这些指令可以一次处理多个数据，在处理结果超过实际处理能力的时候仍能够进行正常处理，并且在软件的配合下，可以得到更强的处理性能。使用 MMX 指令集的好处就是所使用的操作系统可以在不做任何改变的情况下执行 MMX 指令，但是，MMX 指令集的问题也是比较明显的，MMX 指令集不能与 X86 的浮点运算指令同时执行，必须做密集式的交错切换才可以正常执行。但是这样一来，就会造成整个系统运行效率的下降。

2）SSE

SSE 是 Streaming SIMD Extension（SIMD 扩展指令集）的缩写，而其中 SIMD 的含意为 Single Istruction Multiple Data（单指令多数据），所以 SSE 指令集也叫单指令多数据流扩展。该指令集最先运用于 Intel 的 Pentium III 系列处理器，其实在 Pentium III 推出之前，Intel 方面就已经泄露过关于 KNI（Katmai New Instruction）指令集的消息。这个 KNI 指令集也就是 SSE 指令集的前身，当时人们将该指令集称为 MMX2 指令集，但是 Intel 方面却从没有发布有关 MMX2 指令集的消息。最后在 Intel 推出 Pentium III 处理器的时候，SSE 指令集也终于水落石出。SSE 指令集是为提高处理器浮点性能而开发的扩展指令集，它共有 70 条指令，其中包含提高 3D 图形运算效率的 50 条 SIMD 浮点运算指令、12 条 MMX 整数运算增强指令、8 条优化内存中的连续数据块传输指令。理论上这些指令对当时流行的图像处理、浮点运算、3D 运算、多媒体处理等众多多媒体的应用能力起到全面提升的作用。

SSE 指令与 AMD 公司的 3DNow!指令彼此互不兼容，但 SSE 包含了 3DNow!中的绝大部分功能，只是实现的方法不同而已。SSE 也向下兼容 MMX 指令，它可以通过 SIMD 和单时钟周期并行处理多个浮点数据来有效地提高浮点运算速度。

3）3DNow!

3DNow!指令集是由 AMD 公司推出的，该指令集应该是在 SSE 指令之前推出的，被广泛运用于 AMD 的 K6、K6-2 和 K7 系列处理器上，拥有 21 条扩展指令集。在整体上 3DNow! 的 SSE 非常相似，它们都拥有 8 个新的寄存器，但是 3DNow!是 64 位的，而 SSE 是 128 位的。所以 3Dnow!只能存储两个浮点数据，而不是 4 个。但是它和 SSE 的侧重点有所不同，3DNow!指令集主要针对三维建模、坐标变换和效果渲染等 3D 数据的处理，在相应的软件配合下，可以大幅度提高处理器的 3D 处理性能。AMD 公司后来又在 Athlon 系列处理器上开发了新的 Enhanced 3DNow!指令集，新的增强指令数达到了 52 个，以致目前最为流行的 Athlon 64 系列处理器都还支持 3Dnow!指令。

4）SSE 2

在 Pentium III 发布的时候，SSE 指令集就已经集成在了处理器的内部，但因为各种原因

一直没有得到充分的发展。直到 Pentium 4 发布之后，开发人员看到使用 SSE 指令之后，程序执行性能将得到极大的提升，于是 Intel 又在 SSE 的基础上推出了更先进的 SSE 2 指令集。SSE 2 包含了 144 条指令，由两个部分组成：SSE、MMX。SSE 部分主要负责处理浮点数，而 MMX 部分则专门计算整数。SSE 2 的寄存器容量是 MMX 寄存器的 2 倍，寄存器存储数据也增加了 2 倍。在指令处理速度保持不变的情况下，通过 SSE 2 优化后的程序和软件运行速度也能够提高 2 倍。由于 SSE 2 指令集与 MMX 指令集兼容，因此被 MMX 优化过的程序很容易被 SSE 2 再进行更深层次的优化，达到更好的运行效果。SSE 2 对于处理器的性能的提升是十分明显的，虽然在同频率的情况下，Pentium 4 的性能不如 Athlon XP，但由于 Athlon XP 不支持 SSE 2，所以经过 SSE 2 优化后的程序 Pentium 4 的运行速度要明显高于 Athlon XP。而 AMD 方面也注意到了这一情况，在随后的 K-8 系列处理器中，都加入了 SSE 2 指令集。

5）SSE 3

SSE 3 指令集是 Intel 公司在 SSE 2 指令集的基础上发展起来的。相比于 SSE 2，SSE 3 在 SSE 2 的基础上又增加了 13 个额外的 SIMD 指令。SSE 3 中的 13 个新指令的主要目的是改进线程同步和特定应用程序领域，例如媒体和游戏。它共划分为 5 个应用层，分别为数据传输命令、数据处理命令、特殊处理命令、优化命令、超线程性能增强 5 个部分，其中超线程性能增强是一种全新的指令集，它可以提升处理器的超线程的处理能力，大大简化了超线程的数据处理过程，使处理器能够更加快速地进行并行数据处理。这些新增指令强化了处理器在浮点转换至整数、复杂算法、视频编码、SIMD 浮点寄存器操作及线程同步等 5 个方面的表现，最终达到提升多媒体和游戏性能的目的。

6）SSE 4

SSE 4（Streaming SIMD Extensions 4）是英特尔自从 SSE 2 之后对 ISA 扩展指令集最大的一次升级扩展。新指令集增强了从多媒体应用到高性能计算应用领域的性能，同时还利用一些专用电路实现对特定应用加速。从指令数目上看，SSE 4 的 47 条指令增加了的指令改进了整数和浮点操作，支持 DWORD 和 QWORD 操作，新的单精度 FP 操作、快速寄存器操作、面向性能优化的内存操作等，包括了图形、图像、数据装载各方面的革新，因此称其为 SSE 2 以来最大的指令集变动也是不为过的。利用支持 SSE 4 指令集的编译器编译之后，包括图形/图像处理、视频处理、2D/3D 创作、多媒体、游戏、内存敏感负载、高性能计算等应用都会受益。

SSE 4 指令集按照目前的资料将分为两个版本：4.1 和 4.2。SSE 4.1 版本随着 45nm Penryn 发布，而 SSE 4.2 版本将会随着 Penryn 的下一代 Nehalem 发布，4.1 将包括 47 条指令，4.2 将包括 7 条指令。

上面介绍的基本上就是 Intel 和 AMD 公司在 X86 架构处理器上主要的扩展指令集，虽然它们对处理器的性能提升有着一定程度的帮助，但是由于受到 IA-32 体系的限制，X86 架构基本上不会再有具有革命性意义的指令集出现，而双方都已经把重心转向了 64 位体系架构的处理器指令集开发上。

7）AMD64

AMD 的 athlon 64 系列处理器的 64 位技术是在 X86 指令集的基础上加入了 X86-64 的 64 位扩展 X86 指令集，这就使得 athlon 64 系列处理器可兼容原来的 32 位的 X86 软件，并同时支持 X86-64 的扩展 64 位计算，并且具有 64 位的寻址能力，使得它成为真正的 64 位 X86 构

架处理器。在采用 X86-64 架构的 Athlon 64 处理器中，X86-64 指令集中新增了几组处理器寄存器，它能够提供更加快速的执行效率。寄存器是处理器用来创建和存储 CPU 运算结果和其他运算结果的地方，标准的 X86 构架中包括 8 组通用寄存器，而在 AMD 的 X86-64 架构中又增加了 8 组，将通过寄存器的数目提高到了 16 组。在这基础之上，X86-64 指令集还另外增加了 8 组 128 位的 XMM 寄存器，也叫做 SSE 寄存器。

它能够给单指令多数据流技术（SIMD）运算提供更多的存储空间，这些 128 位的寄存器能够提供在矢量和标量计算模式下进行 128 位双精度处理，这也为 3D 数据处理、矢量分析和虚拟技术提供了良好的硬件基础。由于提供了更多的寄存器，按照 X86-64 标准生产的处理器可以更有效率地处理数据，在一个时钟周期内能够传输更多的信息。

8）EM64T

EM64T（Extended Memory 64 Technology）也就是 Intel 公司开发的 64 位内存扩展技术。它实际上就是 Intel IA-32 构架体系的扩展，即 IA-32E（Intel Architectur-32 Extension）。Intel 的 IA-32 处理器通过加入 EM64T 技术便可在兼容 IA-32 软件的情况下，允许软件程序利用更多的内存地址空间，并且允许程序进行 32 位线性地址写入。Intel 的 EM64T 所强调的是 32 位技术与 64 位技术的兼容性，为采用 EM64T 的处理器增加了 8 个 64 位通用寄存器，并将原有的 32 位通用寄存器全部扩展为 64 位，这样也提高了处理器的整数运算能力。另外增加的 8 个 SEE 寄存器也提供了对 SSE、SSE 2 和 SSE 3 指令的支持。

目前 Intel 的主流桌面处理器共有两个模式，传统的 IA-32 模式和 IA-32E 模式。在可支持 EM64T 指令的处理器内有一个扩展功能激活寄存器（Extended Feature Enable Register）称为 IA-32_EFER 的部件，它控制处理器的 EM64T 是否被激活。在一般情况下，处理器会作为一个标准的 IA-32 处理器，在运行 64 位的软件或程序时，EM64T 就会被激活，那么处理器便会在 IA-32E 扩展模式下运行。

9）最新发展

在 2008 年 4 月份的最新 2.28 版本“CPUID 规范”文档中，AMD 公布了 SSE 3、SSE 4.1、SSE 5 这 3 种指令集的 CPUID。CPUID 指令是获得微处理器信息的汇编指令，可以得到处理器的类型、型号、制造商、商标、序列号、频率、缓存等一系列相关信息和规格资料。根据这份文档，SSS3 和 SSE 4.1 的 CPUID 都是“**Fn0000_0001_ECX**”，SSE 5 则是“**Fn8000_0001_ECX**”。当然，这些字符串对普通消费者意义不大，主要是提供给开发人员，比如 CPU-Z 就需要它们来识别处理器。SSE 3 是Intel从 65nm Conroe Core 2 处理器开始引入的扩展指令集，SSE 4.1 则是 Intel 在 45nm Penryn Core 2 系列中最新增加的，而 SSE 5 是 AMD 在去年 8 月底宣布的自己的 128bit 指令集，共有 170 条指令，将在 K10 之后的下一代 Bulldozer 核心架构中实现。Intel 曾经表示无意支持 AMD 的 SSE 5。

以上分析了主要的多媒体汇编指令的基本应用和特点，图 12-1 列出了 SSE 汇编指令集合的关系，包括最早的 SSE 到即将推出的 SSE 5。

10）CPU-Z

CPU-Z 是一款基于 CPID SDK 开发的免费软件，能够检查当前电脑的 CPU 的处理器信息、缓存信息、主板信息和内存信息等。作者所用的电脑是 Pentium 双核 E2410，CPU 详细信息如图 12-2 所示。缓存中分为一级数据、一级指令和二级缓存，DM642 的 CPU 也是分为两级缓存 L1D/L1P 和 L2。只是 CPU 速度没有那么高，为 600MHz。但是 DM642 的 CPU 支持 8 条指令并行，这样就相当于 600×8＝4.8GB 主频，可见 DSP 的处理能力之高。

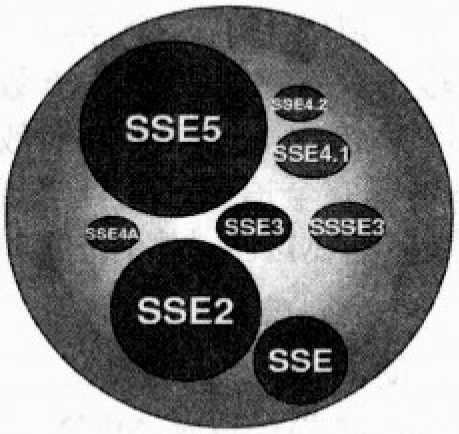


图 12-1　SSE 多媒体汇编指令集合

图 12-2　CPU-Z 检测 CPU 信息

12.2　MMX/SSE 2指令剖析

由于最新的指令都是在原来的基础上新增的，所以这里主要介绍最早的多媒体汇编指令 MMX 和使用得最多的 SSE/SSE 2，其他如 SSE 3/SSE 4 可参考 Intel 的 CPU 指令手册。

12.2.1　MMX媒体扩展指令

Intel 的 MMX 技术是对 Intel 体系结构（IA）指令集的扩展。该技术使用了单指令多数据技术（SIMD），以并行方式处理多个数据元素，从而提高了多媒体和通信软件的运行速度。MMX 指令集包含了 57 条新指令和一个新的 64 位 4 字数据类型。这种新的 64 位数据保持了可供 MMX 指令操作的成组数据值，如图 12-3 所示。

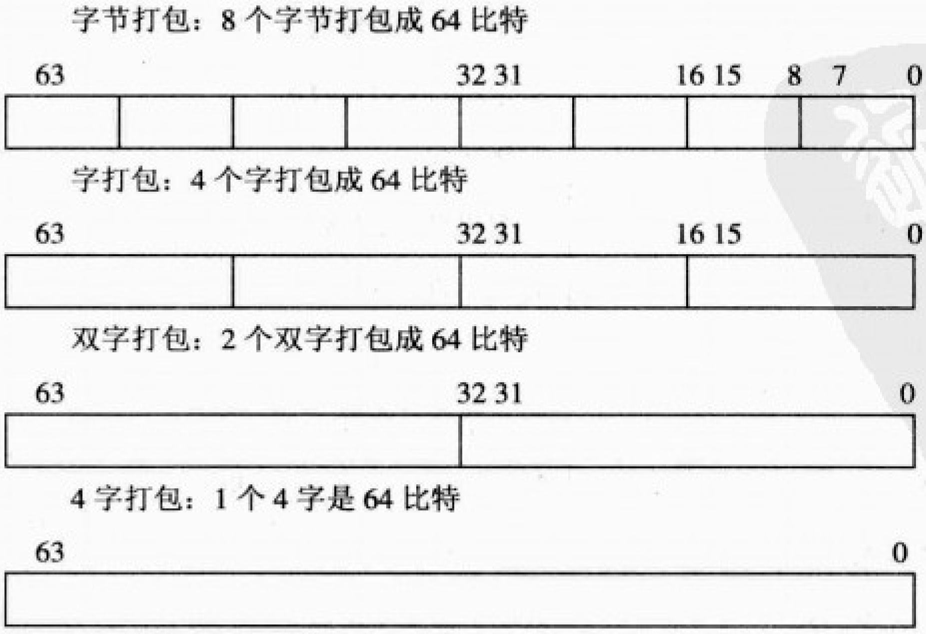


图 12-3　新的数据类型

这样一条 MMX 指令就能够同时处理 8/4/2 个数据单元，这就是所谓的“单指令多数据 SIMD 结构”。这种结构是 MMX 技术把机器性能提高的最根本因素。为了方便使用 64 位打包整形数据，MMX 技术增加了 8 个 64 位的 MMX 寄存器（MM0～MM7），只有 MMX 指令可以使用 MMX 寄存器，每个寄存器可按名称直接访问。

值得一提的是，MMX 寄存器是随机存取的，但实际上是借用了 8 个浮点数据寄存器实现的。浮点处理单元 FPU 有 8 个浮点寄存器 FPR 以堆栈方式存取，每个浮点数据寄存器有 80 位，高 16 位用于指数和符号，低 64 位用于有效数字。MMX 将其 64 位有效数字部分用做随机存取的 64 位的 MMX 寄存器。MMX 技术对操作系统完全透明，并且与现有的、基于 Intel 体系结构的软件 100%兼容，所有应用软件可继续在具有 MMX 技术的处理器上正常运行。

MMX 技术提高了很多应用程序的执行性能，例如活动图像、视频会议、二维图形和三维图形。几乎每一个具有重复性和顺序性整数计算的应用程序都可以从 MMX 技术中受益，对于 8 位、16 位和 32 位数据元素的处理改善了程序的性能。一个 MMX 指令可一次操作 8 个字节，且在一个时钟周期内完成两条指令，也就是说，可在一个时钟周期内处理 16 个数据元素。

另外，为增强性能，MMX 技术为其他功能释放了额外的处理器周期。以前需要其他硬件支持的应用程序现在仅需软件就能运行。更小的处理器占用率给更高程度的并发技术提供了条件，在当今众多的操作系统中这些并发技术得到了应用。在基于 Intel 的分析系统中，某些功能的性能提高了 50%～400%，这种数量级的性能扩展可以在新一代处理器中得到体现。在软件内核中，其速度得到更大的提高，其幅度为原有速度的 3～5 倍。

1. MMX 指令集简介

表 12-1 列出了 MMX 的指令集分类，从功能上可分为 7 组：算术运算指令、比较运算指令、类型转换运算指令、逻辑运算指令、移位运算指令、数据传送指令、状态清除。

表 12-1　MMX 指令集分类

类型	助记符	描述
算术	PADD[B, W, D]	带环绕加
	PADDS[B, W]	带饱和的符号数加
	PADDUS[B, W]	带饱和的无符号数加
	PSUB[B, W, D]	带绕转减
	PSUBS[B, W]	带饱和的符号数减
	PSUBUS[B, W]	带饱和的无符号数减
	PMULHW	打包型字高位乘
	PMULLW	打包型字低位乘
	PMADDMD	打包型字相乘并相加结果对
比较	PCMPEQ[B, W, D]	打包型数相等比较
	PCMPGT[B, W, D]	打包型数大于比较

（续表）

类型	助记符	描述
类型转换	PACKUSWR	字压成字节（无符号数带饱和）
	PACKSS[WB, DW]	字压成字节/双字压成字（有符号数带饱和）
	PUNPCKH[BW, WD, DQ]	把 MMX 寄存器高阶展开
	PUNPCKL[BW, WD, DQ]	把 MMX 寄存器低阶展开
逻辑运算	PAND	按位与
	PANDN	按位与非
	POR	按位或
	PXOR	按位异或
移位	PSLL[W, D, Q]	打包型逻辑左移（位数由立即数或 MMX 寄存器指定）
	PSRL[W, D, Q]	打包型逻辑右移（位数由立即数或 MMX 寄存器指定）
	PSRA[W, D]	打包型算术右移（位数由立即数或 MMX 寄存器指定）
数据传送	MOV[D,Q]	从 MMX 寄存器传入/传出
状态清除	EMMS	清除 MMX 状态

1）算术运算指令

这些指令和普通的算术指令在运算规则上没有什么区别，只不过是在一条指令里同时处理几个数据（Single Instruction and Multi Data，SIMD）而已，但是乘法有些区别。

① 打包数据的加减

PADD（打包数据加）和 PSUB（打包数据减）指令对有符号或无符号数据元素进行加减，按环绕式处理方式将源操作数加到目标操作数上或从目标操作数中减掉源操作数。这些指令支持字节组、字组和双字组。

PADDS（打包数据饱和加）和 PSUBS（打包数据饱和减）指令对有符号或无符号数据元素进行加减，按饱和处理方式将源操作数加到目标操作数上或从目标操作数中减掉源操作数，并按数据类型的取值范围对运算结果进行饱和处理。这些指令支持字节组和字组。

PADDUS（无符号打包数据饱和加）和 PSUBUS（无符号打包数据饱和减）指令对无符号数据元素进行加减，按饱和处理方式将源操作数加到目标操作数上或从目标操作数中减掉源操作数，并按数据类型的取值范围对运算结果进行饱和处理。这些指令支持字节组和字组。

② 打包数据的乘法

打包数据的乘法指令对 16 位无符号操作数按对应关系进行了 4 次乘法，产生了 32 位的中间结果。用户可以选择任何一个 32 位结果的高位或低位部分。

PMULHW（打包数据高位字乘法）和 PMULLW（打包数据低位字乘法）指令将有符号的源操作数和目的操作数的字（Word）相乘，并将每个结果的高 16 位或低 16 位写入目的操作数。

③ 打包数据的乘加

PMADDWD（打包数据相乘并相加）指令将源操作数和目的操作数的有符号字部分相乘，

并将所产生的 4 个 32 位双字的中间结果成对相加，产生两个 32 位双字。

2）比较指令

打包数据的相等、大于比较。PCMPEQ（打包数据相等比较）和 PCMPGT（打包数据大于比较）指令将源操作数和目的操作数对应的数据元素分别进行相等比较或大于比较。这些指令在目的操作数产生 0 或 1 的掩码。逻辑操作可以根据这些掩码选择数据元素。它无须单个分支指令或一组分支指令就能实现有条件的赋值操作，指令不设置标志位。这些指令支持字节组、字组和双字组的数据类型。

3）类型转换指令

字节/字/双字/4 字等之间的转换，即数据的打包、解包。PACKSS（有符号饱和处理模式的类型转换）指令按饱和处理模式，将有符号字组转换成有符号的字节组或将有符号的双字组转换成有符号的字组。

PACKUS（无符号饱和处理模式的类型转换）指令按无符号饱和处理式符号字组转换成无符号的字节组。PUNPCKH（打包数据高位分组）和 PUNPCKL（打包数据低位分组）指令可将字节组转换成字组、将字组转换成双字组或将双字组转换成 4 字组。

4）逻辑运算指令

按位与、与非、或、异或。PAND（逐位逻辑与）、PANDN（逐位逻辑与非）、POR（逐位逻辑或）和 PXOR（逐位逻辑异或）指令在 64 位运算中逐位执行逻辑运算操作。

5）移位指令

打包数据的逻辑左移、右移和算术右移。逻辑左移、逻辑右移和算术右移指令按指定的比特数移动每个数据元素，逻辑左移和逻辑右移也能使 64 位（4 字组）数据作为一个数据块进行移动，辅助数据类型的转换和对齐操作。

PSLL（打包数据逻辑左移）和 PSRL（打包数据逻辑右移）指令执行逻辑左移和逻辑右移，并用 0 填充空出来的高位或低位部分，这些指令支持字组、双字组和 4 字组数据类型。

PSRA（打包数据算术右移）指令执行一个算术右移，并将符号位复制到操作数的高端。该指令支持字组和双字组数据类型。

6）数据传送指令

从寄存器读入或读出。MOVD（传输 32 位）指令将打包的 32 位从内存移到 MMX 寄存器或逆向移动数据，或从整型寄存器中移到 MMX 寄存器或逆向移动数据中。MOVQ（传送 64 位）指令将打包的 64 位从内存移到 MMX 寄存器或逆向移动数据，或在 MMX 寄存器中进行数据移动。

7）MMX 状态清除

EMMS 指令将 MMX 状态清空。该指令必须在一个 MMX 例程结束时执行，用于清除 IA MMX（清空浮点标志字），使下一个例程能够执行浮点运算指令。

除 EMMS 指令外，所有的 MMX 指令操作均涉及两个操作数，即源操作数和目的操作数。右边的操作数为源操作数，左边的操作数为目的操作数。目的操作数也可作为第二个源操作数来使用。指令用结果来覆盖目的操作数。例如，一个双操作数的指令可以按如下方式解码：DEST（左边操作数）←（左边操作数）OP SRC（右边操作数），对所有的 MMX 指令来说（除数据转移指令），源操作数可以位于内存和 MMX 寄存器中。目的操作数寄存于 MMX

寄存器中。对于数据转移指令，源操作数也可以是一个数型寄存器（对 MOVD 指令）或是内存单元（对 MOVD 和 MOVQ 指令）。

MMX 指令功能示例如下：

```
mm0:A7,A6,A5,A4,A3,A2,A1,A0
mm1:B7,B6, B5,B4,B3,B2,B1,B0
punpckhbw mm0,mm1;  // mm0:B7,A7,B6,A6,B5,A5,B4,A4
punpcklbw mm0,mm1 ;  // mm0:B3,A3,B2,A2,B1,A1,B0,A0
```

在上述例程中，已知 mm0 和 mm1 寄存器的 64 位值，上述指令交替取出源操作数和目标操作数的高半部分（Punpckhbw）或低半部分（Punpcklbw），写入目标操作数，即完成数据的打包处理。

2. CPU 寄存器

除上述的 8 个 MMX 寄存器外，CPU 还有自己的寄存器 EAX、EBX、ECX、EDX，除了能够直接访问外，还可分别对其高 16 位和低 16 位进行访问。它们的低 16 位就是把它们前面的 E 去掉，即 EAX 的低 16 位就是 AX。而且它们的低 16 位又可以分别进行 8 位访问，也就是说，AX 还可以再进行分解，即 AX 还可分为 AH（高 8 位），AL（低 8 位）。另外，CPU 还有几个寄存器：ESI、EDI、ESP、EBP。CPU 寄存器及其功能描述如表 12-2 所示。

表 12-2　CPU 寄存器

寄存器	功能描述	寄存器	功能描述
EAX	累加器	ESI	源变址寄存器
EBX	基址寄存器	EDI	目的变址寄存器
ECX	一般用来当做计数器	ESP	堆栈指针寄存器
EDX	数据寄存器	EBP	基址指针寄存器

其中 ESP 和 EBP 一般是针对堆栈而言的。

3. 地址加载指令

1）LEA 指令（Load Effective Address）

LEA 指令把源操作数的地址偏移量传送给 16 位通用寄存器。该指令常用来建立操作所需要的寄存器地址指针，格式为：LEA r,src ;src 代表源操作数，r 代表寄存器。

举例如下。

- LEA BX,[2400H];：将 2400 单元（偏移地址）送给 BX，指令执行后 BX=2400H。
- LEA SI,TABLE;：把 TABLE（偏移地址）送给 SI，执行后 SI 指向 TABLE；该指令和 MOV SI、OFFSET TABLE 的作用一样。

2）LDS（Load Pointer Into DS）

LDS 指令是把一个存放在 4 个存储单元中共为 32 位的目标指针（段地址和偏移量）传送到两个目的寄存器，其中后两个字节（高地址）内容即段地址送到 DS；前 2 个地址（低地址）内容即偏移量送到指令中所出现的寄存器中，举例如下。

- LDS DI,[2130H];把 2132H、2133H 单元中的内容送到 DS；把 2130H、213lH 单元中的内容送到 DI。
- LDS SI,DWORD PTR CS:[BX]:把代码段 BX 指向的 4 个单元内容送到 DS 和 SI，其中[BX+2]、[BX+3]送到 DS，[BX][BX+1];送到 SI，DWORD PTR 是属性运算符，它指出是双字（4 字节）。

3）LES（Load Pointer into ES）

这条指令除把目标段地址送给 ES 外，其他与 LDS 相同，LDS/LES 指令的源操作数都来自于存储器。

12.2.2 MMX程序设计

使用 MMX 技术编程有多种方法。

- 直接编写汇编程序，存为*.asm 文件。
- 在 C++中写嵌入式汇编，使用__asm{ … }编写汇编程序。
- 使用 64 位的数据类型__m64 和一系列 C++函数（Intrinsic 指令）来进行相关的算术和逻辑运算，类似于 DSP 的 intrinsic 指令。

开源的视频编/解码算法如 Xvid、ffmpeg、X264 等核心算法模块就使用了直接编写汇编程序，存为*.asm 文件的方式。这种方法的语句调试由于是在汇编环境下进行的，因此不友好，也不方便。这里主要介绍后两种方法，即使用 MMX 进行程序设计。根据 MSDN 提供的两个例程 MMX8/MMX32 展示如何使用 MMX 进行程序设计。

1. 基本概念

1）包含头文件

所有的 MMX 指令集函数在 emmintrin.h 文件中定义：

```
#include <emmintrin.h>
```

因为程序中用到的 MMX 处理器指令是由编译器决定的，所以它并没有相关的.lib 库文件。

2）__m64 数据类型

这种类型的变量可用做 MMX 指令的操作数，它不能被直接访问。__m64 类型的变量被自动分配为 8 字节的字长。

3）CPU 对 MMX 指令集的支持

如果读者的 CPU 具有 MMX 指令集，就可以使用 Visual Studio 2005 提供的对 MMX 指令集支持的 C++函数库，读者可以查看 MSDN 中的一个 Visual C++ CPUID 的例子，它可以帮读者检测读者的 CPU 是否支持 SSE、MMX 指令集或其他的 CPU 功能。

4）饱和算法（Saturation Arithmetic）和封装模式（Wraparound Mode）

MMX 技术支持一种叫做 Saturation Arithmetic（饱和算法）的计算模式。在饱和模式下，当计算结果发生溢出（上溢或下溢）时，CPU 会自动去掉溢出的部分，使计算结果取该数据类型表示数值的上限值（如果上溢）或下限值（如果下溢）。饱和模式的计算用于对图像的

处理。

下面的例子能够让读者理解饱和模式与封装模式的区别。如果一字节（Byte）类型变量的值为 255，然后将其值加 1（上溢出）。在封装模式下，相加结果为 0（去掉进位）；在饱和模式下，结果为 255。饱和模式用类似的方法来处理下溢出（小于 0，则以 0 作为结果），比如对于一个字节数据类型的数在饱和模式下，1 减 2 的结果为 0（而不是–1）。每一个 MMX 算术指令都有这两种模式：饱和模式和封装模式。这里所要讨论的项目只使用饱和模式下的 MMX 指令。

2．编程实例

以下实例讲解了 MMX 技术在 Visual Studio 2005 下的应用实例，读者可以在 http://www.codeproject.com /KB/recipes/mmxintro/MMX_src.zip 下载示例程序压缩包，网站需要用户注册并登录后才能下载。该压缩包中含有两个项目，这两个项目是基于微软基本类库（MFC）建立的 Visual C++.NET 项目，读者也可以按照下面的讲解建立这两个项目。不过需要注意的是，这个项目是在 VS 2003.NET 平台建立的项目，如果在 VS 2005 上运行，则需要在回掉函数前加地址符&，并纠正编译器提示的其他错误，详细代码见光盘。

MMX8 是一个单文档界面（SDI）的应用程序，用来对每像素 8 位的单色位图进行简单处理，源图像和处理后的图像会在窗口中显示出来。新建的 ATL（活动模板库）类 Cimage 用来从资源中提取图像并在窗体中显示出来。程序要对图像进行两种处理操作：图像颜色反相和改变图像的亮度。每一种处理操作可以用下面几种方法之中的一种来实现。

- 纯 C++代码。
- 使用 C++的 MMX 功能函数的代码。
- 使用 MMX 汇编指令的代码。

对图像进行处理计算的时间会显示在状态栏中。

用纯 C++实现的图像颜色反相函数代码如下。

```
void CImg8Operations::InvertImageCPlusPlus( BYTE* pSource,BYTE* pDest, int
nNumberOfPixels)
{
  for( int i = 0; i < nNumberOfPixels; i++ )//循环所有的像素点
  {
    *pDest++ = 255 - *pSource++;              //反色处理，即像素最大值255减当前像素值
  }
}
```

上述程序就是实现对传入的图像做反色处理，处理过程为像素最大值 255 减去当前像素值，最后返回处理后的目标图像空间指针。

为了查询使用 C++ MMX 指令函数的方法，用户应参考 Intel 软件说明书（Intel Software manuals）中有关 MMX 汇编指令的说明，首先在第一卷的第 8 章介绍 MMX 相关指令的总体，然后在第二卷介绍有关这些 MMX 指令的详细说明，这些说明有一部分涉及了与其特性相关的 C++函数。然后通过这些 MMX 指令对应的 C++函数查找了 MSDN 中与其相关的说明。在 MMX8 示例程序中用到的 MMX 指令和相关的 C++函数如表 12-3 所示。

表 12-3 MMX8 项目用的助记符和 intrinsic 指令

函数说明	助记符	MMX Intrinsic
清空 MMX 状态（禁止与 float 操作冲突）	emms	_mm_empty
8 字节对应的无符号饱和相减	psubusb	_mm_subs_pu8
8 字节对应的无符号饱和相加	paddusb	_mm_adds_pu8

```
void CImg8Operations::InvertImageC_MMX(BYTE* pSource, BYTE* pDest, int nNumber
OfPixels) {
    __int64 i = 0;
    i = ~i;                                    //0xffffffffffffffff
    //一次循环处理8个像素点
    int nLoop = nNumberOfPixels/8;             //循环次数
    __m64* pIn = (__m64*) pSource;             //输入指针
    __m64* pOut = (__m64*) pDest;              //输出指针
    __m64 tmp;                                 //临时变量
    _mm_empty();                               //emms，清空mmx状态
    __m64 n1 = Get_m64(i);                     //获取与i相同值的__m64类型的变量值
    for ( int i = 0; i < nLoop; i++ )      //循环所有像素点，每次处理8个像素点
    {
        tmp = _mm_subs_pu8 (n1 , *pIn); //8字节对应的无符号饱和相加
                                        //即：tmp = n1 - *pIn，每对字节做减法
        *pOut = tmp;
        pIn++;                                 //下一个 8像素
        pOut++;
    }
    _mm_empty();                               //清空MMX状态
}
    /* 获取与in64相同值的__m64类型的值*/
__m64 CImg8Operations::Get_m64(__int64 n) {
    union __m64__m64 {
        __m64 m;
        __int64 i;
    } mi;                                      //mi为联合体类型
    mi.i = n;                                  //int64类型
    return mi.m;                               //返回__m64
}
```

虽然这个函数在非常短的时间内就执行完成了，但记录了如下 3 种方法需要的时间结果。

- 纯 C++代码 27 毫秒。
- 使用 C++的 MMX 指令函数的代码 5 毫秒。
- 使用 MMX 汇编指令的代码 5 毫秒。

上面的图像处理程序 Release 优化编译后执行时，时间更短，分别是 7 毫秒、1 毫秒、1 毫秒。按照如下步骤执行该程序 MMX8，测试程序的 Debug 版本执行时间并查看效果图。

STEP 01 按“F5”键，运行该程序，效果如图 12-4 所示。

STEP 02 选择“Operations”→“Invert”→“C++”命令，使用 C++语言处理反色的效果如图 12-5 所示。

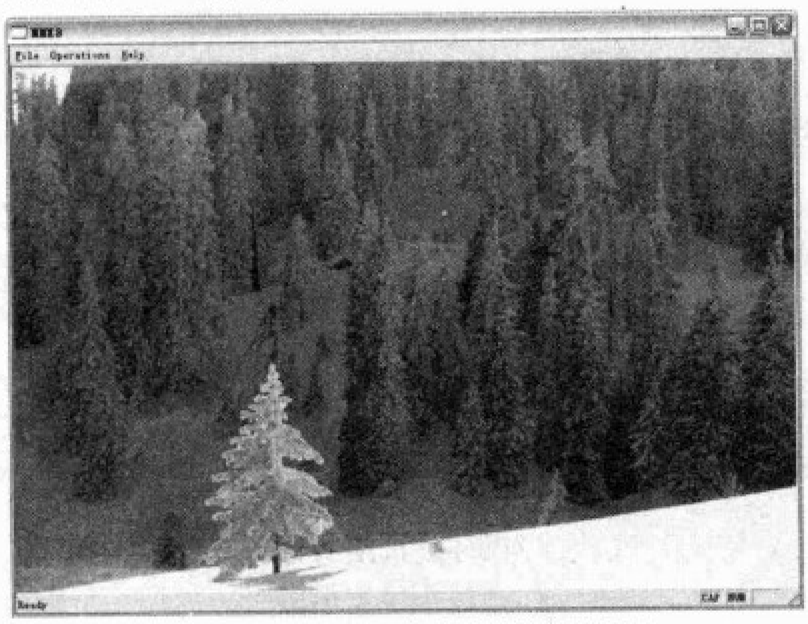


图 12-4　MMX8 应用程序效果图

图 12-5　C++语言处理效果图

STEP 03 选择“Operations”→“Reset”命令重新加载位图，然后选择“Operations”→“Invert”→“C++ With MXX”命令，使用 MMX 的 C++处理效果如图 12-6 所示。

STEP 04 选择“Operations”→“Reset”命令重新加载位图，然后选择“Operations”→“Invert”→“Assembly With MXX”命令，使用 MMX 的汇编处理效果如图 12-7 所示。

图 12-6　使用 MMX 的 C++处理效果图

图 12-7　使用 MMX 的汇编处理效果图

在 MMX8 应用程序的状态栏标出了当前处理方式的时间消耗情况。

而改变图像的亮度采用了最简单的方法：对图像中的每一个像素的颜色值进行加减运算。相对前面的处理函数而言，这样的转换函数有些复杂，因为需要把处理过程分成两种情况，一种是增加像素颜色值，另一种是减少像素颜色值。

用纯 C++函数实现的改变图像亮度的函数，代码如下。

```
void CImg8Operations::ChangeBrightnessCPlusPlus(BYTE* pSource, BYTE* pDest,
                                    int nNumberOfPixels, int nChange)
{
```

```
    //把输入的nChange值嵌位在-255~255范围内
    if ( nChange > 255 )   nChange = 255;
    else  if  ( nChange < -255 )
      nChange = -255;
    //nChange取绝对值赋给b
    BYTE b = (BYTE) abs(nChange);
    int i, n;
    if ( nChange > 0 ){                                  //增加像素颜色值
         for ( i = 0; i < nNumberOfPixels; i++ )         //循环处理所有像素
          {    n =     (int)(*pSource++ + b);             //把每个像素值加b
             if ( n > 255 )    n = 255;                  //嵌位到0~255
             *pDest++ = (BYTE) n;                        //保存
         }
    } else{                                              //减少像素颜色值
 for ( i = 0; i < nNumberOfPixels; i++ ) {               //循环处理所有像素
             n = (int)(*pSource++ - b);                  //把每个像素值减b
             if ( n < 0 )    n = 0;                      //嵌位到0~255
             *pDest++ = (BYTE) n;                        //存
         }
     }
 }
```

上述代码首先把要改变的亮度值嵌位在−255～255 范围内，并取绝对值赋给 b。若要改变的值大于 0，则把图像的每个像素值加 b。若小于 0，则把图像的每个像素值减 b。最后把改变后的像素保存。

用 Visual C++ 2005 的 MMX 指令函数实现的改变图像亮度函数，代码如下。

```
void CImg8Operations::ChangeBrightnessC_MMX(BYTE* pSource, BYTE* pDest,
                                          int nNumberOfPixels, int nChange)
{
  //把输入的nChange值嵌位在-255~255范围内
if ( nChange > 255 )   nChange = 255;
   else if ( nChange < -255 )
       nChange = -255;
   //nChange取绝对值赋给b
   BYTE b = (BYTE) abs(nChange);
  __int64 c = b;
   //把b组合打包成64位。8字节，每字节均为b
   for ( int i = 1; i <= 7; i++ ) {  c = c << 8;  c |= b;  }
   //在一次循环中处理8个像素
   int nNumberOfLoops = nNumberOfPixels / 8;  //循环次数
   __m64* pIn = (__m64*) pSource;             //输入的字节数组
   __m64* pOut = (__m64*) pDest;              //输出的字节数组
   __m64 tmp;                                 //临时工作变量
   _mm_empty();                               //执行MMX指令：清空mmx状态
   __m64 nChange64 = Get_m64(c)               //把int64类型的c转换成__m64类型
   if ( nChange > 0 ) {
```

```
        for ( i = 0; i < nNumberOfLoops; i++ )
        {
            tmp = _mm_adds_pu8(*pIn, nChange64);   //饱和模式下的无符号加法
                                    //对每一个字节执行操作：tmp = *pIn + nChange64
            *pOut = tmp;
            pIn++;                                 //取下面8个像素
            pOut++;
        }
    }else{
        for ( i = 0; i < nNumberOfLoops; i++ )
    {
            tmp = _mm_subs_pu8(*pIn, nChange64);   //饱和模式下的无符号减法
                                    //对每一个字节执行操作：tmp = *pIn - nChange64
            *pOut = tmp;
            pIn++;                                 //取下面8个像素
            pOut++;
    }
    }
    _mm_empty();                                   //执行MMX指令：清空mmx状态
    }
```

上述程序实现使用 MMX 的汇编指令在 C++环境下的应用。注意在开始结束时清空 MMX 状态，以确保 MMX 正常运行。

用 Visual C++ 2005 的 MMX 嵌入式汇编编程实现的改变图像亮度函数，代码如下。

```
    //用MMX汇编改变亮度
    void CImg8Operations::ChangeBrightnessAssembly_MMX(
        BYTE* pSource,
        BYTE* pDest,
        int nNumberOfPixels,
        int nChange)
    {
        //转换 abs(nChange)的值为无符号字节b
        BYTE b = (BYTE) abs(nChange);
        if ( b < 0 )    b = 0;
        else if ( b > 255 )  b = 255;
        //打包64位，每个字节为b
        __int64 nChange64 = b;
        for ( int i = 1; i <= 7; i++ ) {
            nChange64 = nChange64 << 8;
            nChange64 |= b;
        }
        //一次循环处理8字节
        int nNumberOfLoops = nNumberOfPixels / 8;
        if ( nChange > 0 )                        //加b
        {
            __asm
```

```
            {
                emms
                mov        esi, pSource          //输入指针
                mov        edi, pDest            //输出指针
                mov        ecx, nNumberOfLoops   //循环次数
                movq       mm1, nChange64        //mm1 = nChange64
    start_loop:
                movq       mm0, [esi]            //mm0 = [esi]
                paddusb     mm0, mm1             //无符号带饱和相加
                                                 //每个字节执行mm0 = mm0 + mm1
                movq       [edi], mm0            //[edi] = mm0 (8 pixels)
                add        esi, 8                //指针递增，指向后面的64比特
                add        edi, 8                //指针递增，指向后面的64比特
                dec        ecx
                jnz        start_loop
                emms
            }
        } else {                                 //减b
            __asm {
                emms
                mov        esi, pSource          //输入指针
                mov        edi, pDest            //输出指针
                mov        ecx, nNumberOfLoops   //循环次数
                movq       mm1, nChange64        //mm1 = nChange64
    start_loop1:
                movq       mm0, [esi]            //mm0 = [esi]
                psubusb     mm0, mm1             //无符号饱和相减
                                                 //每个字节执行：mm0 = mm0 - mm1
                movq       [edi], mm0            //[edi] = mm0 (8 pixels)
                add        esi, 8                //指针递增，指向后面的64比特
                add        edi, 8                //指针递增，指向后面的64比特
                dec        ecx
                jnz        start_loop1
                emms
            }
        }
    }
```

上述程序与前一个的实现实质基本相同，执行效率略有区别。注意使用这种嵌入式汇编时，关键字__asm 的使用。

注意：

参数 nChange 的符号每次调用函数时在循环体外只检查一次，而不是放在循环体内，那样会被检查成千上万次。

下面所示为以上 3 种方法处理图像花费的时间。

- 纯 C++代码 46 毫秒。
- 使用 C++的 MMX 指令函数的代码 5 毫秒。
- 使用 MMX 汇编指令的代码 1 毫秒。

现在分别运行这 3 种处理方式，查看应用程序 Debug 版的图像处理效果和不同处理方式的时间消耗情况。

STEP 01 按“F5”键运行该程序，效果如图 12-8 所示。

STEP 02 选择“Operations”→“Change Brightness”→“C++”命令，使用 C++语言处理反色的效果如 12-9 所示。

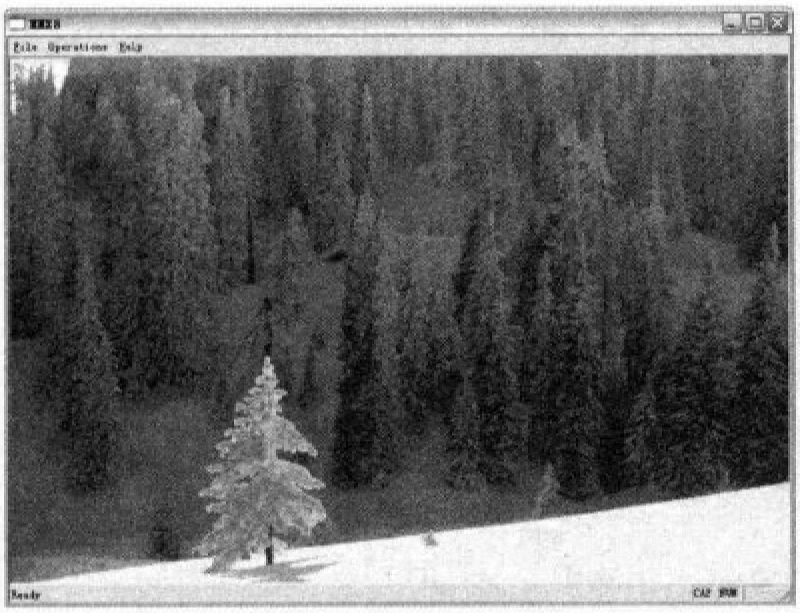


图 12-8 MMX8 应用程序效果图

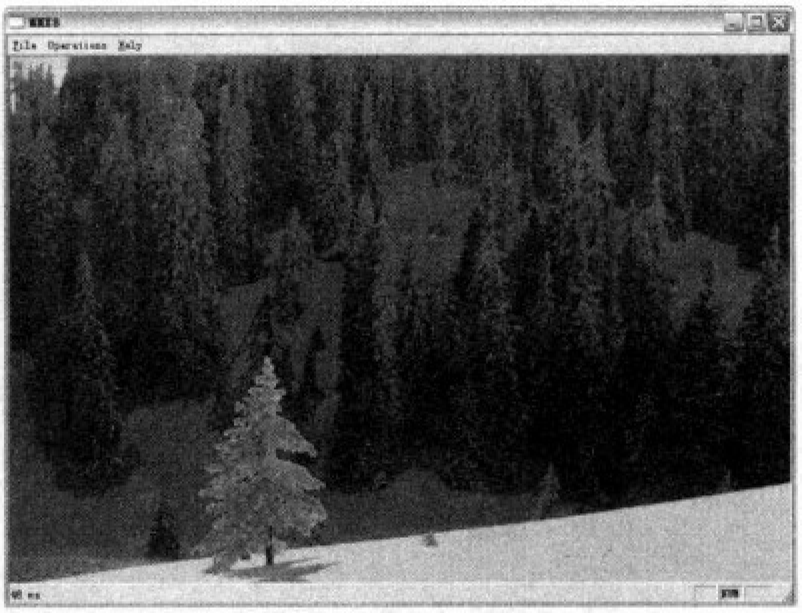


图 12-9 C++语言处理效果图

STEP 03 选择“Operations”→“Reset”命令重新加载位图，然后选择“Operations”→“Change Brightness”→“C++ With MXX”命令，使用 MMX 的 C++处理效果如图 12-10 所示。

STEP 04 选择“Operations”→“Reset”命令钮重新加载位图，然后选择“Operations”→“Change Brightness”→“Assembly With MXX”命令，使用 MMX 的汇编处理效果如图 12-11 所示。

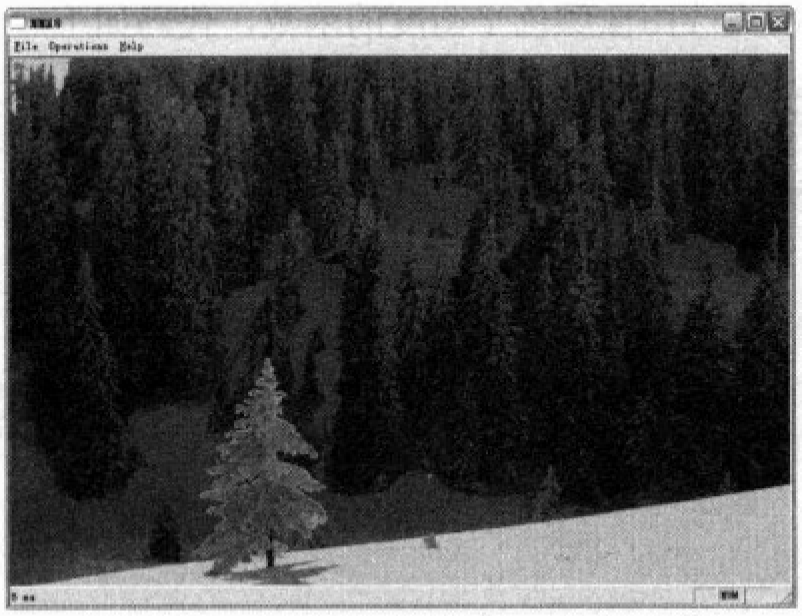


图 12-10 使用 MMX 的 C++处理效果图

图 12-11 使用 MMX 的汇编处理效果图

在 MMX8 应用程序的状态栏标出了当前处理方式的时间消耗情况。

MMX32 项目可对 32 位像素的 RGB 图像进行处理，进行的图像处理工作是图像颜色反

相操作和更改图像颜色的平衡度（将像素点的每一种颜色乘以一定的值）操作。这里通过介绍更改图像颜色的平衡度处理，用 3 种方法来实现。

MMX 的乘法实现起来比加减法复杂得多，因为乘法运算通常得出的结果的位数不再是以前位数的大小。比如，如果乘法的操作数有一字节（8 位的 Byte）大小，那么结果会到一个字（16 位的 Word）大小。这需要额外的转换，并且使用 MMX 汇编指令和 C++代码进行图像转换花费时间的差别不是很大（时间差为 5%～10%）。

用纯 C++函数实现更改图像平衡度，代码如下。

```
void CImg32Operations::ColorsCPlusPlus( BYTE* pSource,  BYTE* pDest,
                              int nNumberOfPixels,  float fRedCoefficient,
                              float fGreenCoefficient,  float fBlueCoefficient)
{
     int nRed = (int)(fRedCoefficient * 256.0f);      //红色分量平衡度左移
     int nGreen = (int)(fGreenCoefficient * 256.0f);  //绿色分量平衡度左移
     int nBlue = (int)(fBlueCoefficient * 256.0f);    //蓝色分量平衡度左移
     for ( int i = 0; i < nNumberOfPixels; i++ )      //循环所有像素点（占32位）
     {
     *pDest++ = (BYTE) ( ((int)(*pSource++ * nBlue)) >> 8 );
     //每个像素的R乘以平衡度，然后右移
     *pDest++ = (BYTE) ( ((int)(*pSource++ * nGreen)) >> 8 );
     //每个像素的G乘以平衡度，然后右移
     *pDest++ = (BYTE) ( ((int)(*pSource++ * nRed)) >> 8 );
     //每个像素的B乘以平衡度，然后右移
         pDest++;                                     //跳过alpha分量
         pSource++;                                   //跳过alpha分量
     }
}
```

上述程序首先对 3 个浮点类型的平衡度变量左移 8 位，接着对所有 32 位的像素点进行处理，即将原像素值乘以平衡度，然后右移，最后跳过 alpha 分量。

用 Visual C++ 2005 的 MMX 指令函数实现的更改图像颜色平衡度的函数，代码如下。

```
void CImg32Operations::ColorsC_MMX( BYTE* pSource, BYTE* pDest,
                              int nNumberOfPixels,  float fRedCoefficient,
                              float fGreenCoefficient, float fBlueCoefficient)
{
  int nRed = (int)(fRedCoefficient * 256.0f);         //红色分量平衡度左移
  int nGreen = (int)(fGreenCoefficient * 256.0f);     //绿色分量平衡度左移
  int nBlue = (int)(fBlueCoefficient * 256.0f);       //蓝色分量平衡度左移
  // 平衡系数打包成64位
  __int64 c = 0;
  c = nRed;
  c = c << 16;
  c |= nGreen;
  c = c << 16;
  c |= nBlue;
  __m64 nNull = _m_from_int(0);              //int64类型的0转换成__m64类型
```

```
    __m64 tmp = _m_from_int(0);                  //临时工作临时变量初始化为0
    _mm_empty();                                 //清空MMX状态
    __m64 nCoeff = Get_m64(c);                   //c变量转换成__m64类型
    DWORD* pIn = (DWORD*) pSource;               //输入指针，指向32位
    DWORD* pOut = (DWORD*) pDest;                //输出指针，指向32位
    for ( int i = 0; i < nNumberOfPixels; i++ )     {
        tmp = _m_from_int(*pIn);                 //tmp = *pIn (在tmp的低32位写入数据)
        tmp = _mm_unpacklo_pi8(tmp, nNull );
        //将tmp中低位的4个字节转化为字，字的高位用nNull中对应位上的值填充
        tmp = _mm_mullo_pi16 (tmp , nCoeff);
        //将tmp中的每一个字相乘，将相乘结果的，高位送到nCoeff，在tmp中只保留每个结果的低位
        tmp = _mm_srli_pi16 (tmp , 8);          //将tmp中的每一个字右移8位，/256
        tmp = _mm_packs_pu16 (tmp, nNull);
        //使用饱和模式将tmp中的结果做如下处理：/将tmp中的4个字转化为4字节，并将这4字节
    写到tmp中的低32位中，同时，将nNull中的4个字转化为4字节，并将这4字节写到tmp的
    高32位中
        *pOut = _m_to_int(tmp); //*pOut = tmp (将tmp低32位的数据放入pOut数组中)
        pIn++;                       //修改指针
        pOut++;                      //修改指针
    }
    _mm_empty();                     //清空MMX状态
}
```

上述代码首先取得整数类型的平衡度系数，接着把 3 个平衡度系数打包到__m64 类型的数据中，循环处理所有像素点。

用 MMX 汇编实现 32 位彩色图像颜色转换，代码如下。

```
void CImg32Operations::ColorsAssembly_MMX( BYTE* pSource,  BYTE* pDest,
                              int nNumberOfPixels,  float fRedCoefficient,
                              float fGreenCoefficient, float fBlueCoefficient)
{
    int nRed = (int)(fRedCoefficient * 256.0f);
    int nGreen = (int)(fGreenCoefficient * 256.0f);
    int nBlue = (int)(fBlueCoefficient * 256.0f);
    __int64 nCoeff = 0;
    __int64 nNull = 0;
    //平衡度系数打包
    nCoeff = nRed;
    nCoeff = nCoeff << 16;
    nCoeff |= nGreen;
    nCoeff = nCoeff << 16;
    nCoeff |= nBlue;
    __asm
    {
        emms                                      //清空MMX状态
        mov       esi, pSource                    //输入指针
        mov       edi, pDest                      //输出指针
        mov       ecx, nNumberOfPixels            //循环次数
```

```
        //movq指令从源地址空间复制4字到目标地址
        movq        mm1, nNull     //mm1=0
        movq        mm2, nCoeff                    //打包后的平衡度系数
    start_loop:
        movd        mm0, [esi]          //把ARGB32位装载到MMX0
        punpcklbw mm0, mm1              //打包mm0和mm1的低32位，交替取mm0和mm1的低位
        pmullw      mm0, mm2            //mm0和mm2的每个字相乘
                                        //把结果的低位字赋给mm0
        psrlw       mm0, 8              //mm0的每个字右移8位
        packuswb    mm0, mm1            //有符号数WORD带饱和压缩成无符号BYTE
        movd        [edi], mm0          //把mm0的低32位数据写到edi指向的空间
        add         esi, 4              //修改输入指针
        add         edi, 4              //修改输出指针
        dec         ecx                 //循环计数减1
        jnz         start_loop          //判断循环
        emms                            //清空MMX状态
      }
    }
```

与上一个代码中的 MMX 指令函数的实现过程基本相同。

上面列出了部分 MMX8/MMX32 项目的处理函数，其他模块见光盘上的工程 chapter12\MMX 目录。

现在分别运行这 3 种处理方式，查看应用程序 MMX32 的 Release 版的图像处理效果和不同处理方式的时间消耗情况。

STEP 01 按“F5”键运行该程序，效果如图 12-12 所示。

STEP 02 选择“Operations”→“Change Brightness”→“C++”命令，使用 C++语言处理反色的效果如图 12-13 所示。

STEP 03 选择“Operations”→“Reset”命令重新加载位图，然后选择“Operations”→“Change Brightness”→“C++ With MXX”命令，使用 MMX 的 C++处理效果如图 12-14 所示。

STEP 04 选择“Operations”→“Reset”命令重新加载位图，然后选择“Operations”→“Change Brightness”→“Assembly With MXX”命令，使用 MMX 的汇编处理效果如图 12-15 所示。

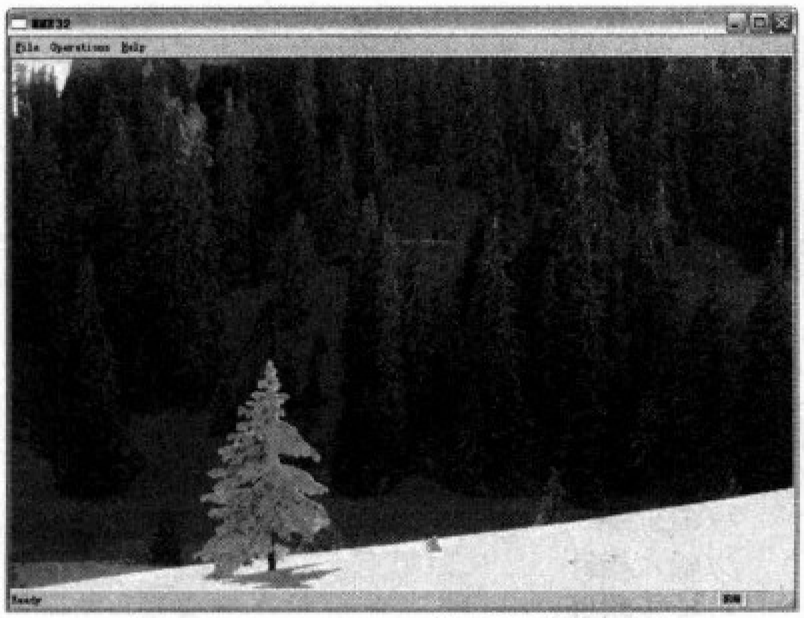


图 12-12　MMX8 应用程序效果图

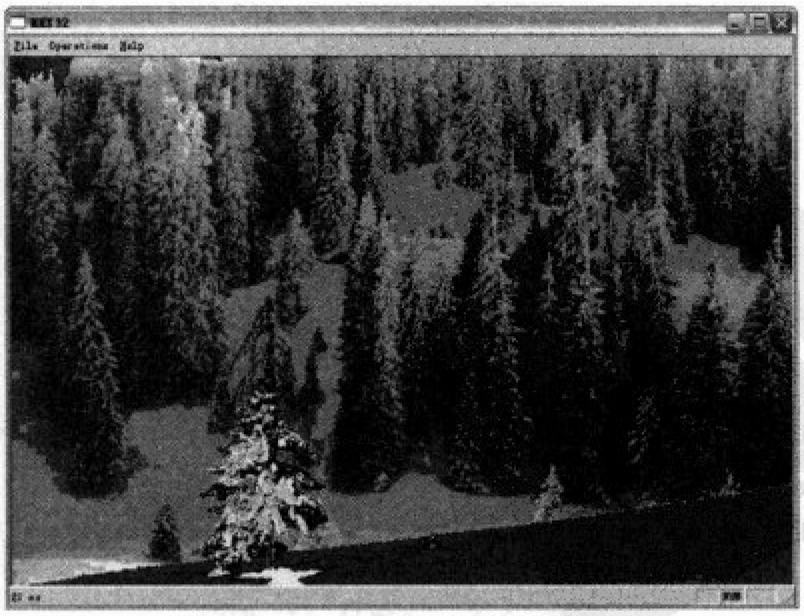


图 12-13　C++语言处理效果图

图 12-14 使用 MMX 的 C++处理效果图

图 12-15 使用 MMX 的汇编处理效果图

在 MMX32 应用程序的状态栏标出了当前处理方式的时间消耗情况。

12.2.3 SSE/SSE 2媒体扩展指令

上面介绍了 MMX 指令的多媒体处理过程、MMX 寄存器的使用注意事项等。微软看到 MMX 指令大大地提高了程序处理速度，于是相继推出了基于 MMX 的新的汇编指令 SSE、SSE 2，把寄存器从 64 位扩展到 128 位，并且又增加了新的媒体处理汇编指令。

1. SSE/SSE 2 数据结构

SSE 技术支持的主要数据类型是打包的单精度浮点操作数（Packed Single-Precision Floating-point）。它是将 4 个互相独立的 32 位单精度（Single-Precision，SP）浮点数据打包在一个 128 位的数据中。32 位单精度数据格式符合 IEEE 754 标准，SSE 技术提供了 8 个 128 位的 SIMD 浮点数据寄存器。每个 SIMD 浮点数据寄存器都可以直接存取，寄存器名为 XMM0～XMM7，它用于存放数据而不能用于寻址存储器，SSE 技术还提供了一个新的控制/状态寄存器 MXCSR。

SSE 2 指令系统包括 IA-32 微处理器原有的 32 位通用寄存器、64 位 MMX 寄存器、128 位 XMM 寄存器，还包括 32 位的标志寄存器 EFLAGS 和浮点状态/控制寄存器 MXCSR；但并没有引入新的寄存器和指令执行状态。它主要利用 XMM 寄存器新增了一种 128 位的打包的双精度浮点数据和 4 种 128 位 SIMD 整型数据类型。

- 打包的双精度浮点数（Packed Double-precision Floating-point）：这个 128 位数据类型由两个符合 IEEE 标准的 64 位双精度浮点数组成，打包成一个双 4 字数据。
- 128 位打包的整数（128bit Packed Integer）：这 4 种 128 位打包整型数据可以包含 16 字节整数、8 字整数、4 个双字整数或 2 个 4 字整数。

SSE 2 指令系统中最主要的指令就是针对 128 位和 64 位操作模式的打包的双精度浮点指令。另外还有 64 位和 128 位 SIMD 整数指令、MMX 和 SSE 技术的 128 位扩展指令、高速缓存控制和指令排序指令。

2. SSE/SSE 2 指令

SSE 指令集有 70 条指令，它们可以分成 3 组。

1）50 条 SIMD 浮点指令

SSE 技术中的 50 条 SIMD 浮点指令是 SSE 指令系统的主要指令，也是 Pentium III 处理器性能提高的一个关键。它又可以分成几组：数据传送指令、算术运算指令、逻辑运算指令、比较指令、类型转换指令、组合指令、状态管理指令。

2）12 条 SIMD 整数指令

SSE 指令集中有 12 条 SIMD 整数指令。这是为了增强和完善 MMX 指令系统而新增加的指令。

3）8 条高速缓冲存储器优化处理指令

为了更好地控制 Cache 的操作，提高程序运行性能，SSE 技术针对 Pentium III 设计了 8 条高速缓存的优化处理指令。

SSE 2 指令系统中最主要的指令就是针对 128 位和 64 位操作模式的紧缩双精度浮点指令。另外还有 64 位和 128 位的 SIMD 整数指令、MMX 和 SSE 技术的 128 位扩展指令、高速缓存控制和指令排序指令。

- SSE 2 浮点指令：SSE 2 浮点指令分成多组，有 SSE 2 的传送、算术运算、逻辑运算、比较、组合、转换指令。
- SSE 2 扩展指令：SSE 2 技术除具有双精度浮点指令外，还在原来的 MMX 和 SSE 技术基础上补充了 SIMD 扩展整数指令、高速缓存控制和指令排序指令。

3. SSE 开发应用

Intel 公司的单指令多数据流式扩展（Streaming SIMD Extensions，SSE）技术能够有效地增强 CPU 浮点运算的能力。Visual Studio 2005 提供了对 SSE 指令集的编程支持，从而允许用户在 C++代码中不用编写汇编代码就可直接使用 SSE 指令。

SIMD（Single-instruction，Multiple-data）是一种使用单道指令处理多道数据流的 CPU 执行模式，即在一个 CPU 指令执行周期内用一道指令完成处理多个数据的操作。考虑一下下面这个任务，计算一个很长的浮点型数组中每一个元素的平方根。实现这个任务的算法可以这样写：

```
for each f in array        //对数组中的每一个元素计算它的平方根
f = sqrt(f)
```

为了了解实现的细节，把上面的代码这样改写：

```
for each f in array  {
    把f从内存加载到浮点寄存器
    计算平方根
    再把计算结果从寄存器中取出放入内存
}
```

具有 Intel SSE 指令集支持的处理器有 8 个 128 位的寄存器，每一个寄存器可以存放 4 个

（32 位）单精度的浮点数。SSE 同时提供了一个指令集，其中的指令可以允许把浮点数加载到这些 128 位的寄存器中，这些数就可以在这些寄存器中进行算术逻辑运算，然后把结果放回内存。采用 SSE 技术后，算法可以写成下面的样子：

```
for each 4 members in array//对数组中的每4个元素
{
    把数组中的这4个数加载到一个128位的SSE寄存器中
    在一个CPU指令执行周期中完成计算这4个数的平方根的操作
    把所得的4个结果取出写入内存
}
```

C++编程人员在使用 SSE 指令函数编程时不必关心这些 128 位的寄存器，可以直接使用 128 位的数据类型“__m128”和一系列 C++函数来实现这些算术和逻辑操作，而决定程序使用哪个 SSE 寄存器及代码优化是 C++编译器的任务。当需要对很长的浮点数数组中的元素进行处理的时候，SSE 技术确实是一种很有效的方法。

12.2.4 SSE程序设计详细解析

以上对 SSE/SSE 2 的数据结构、指令集合及开发应用注意事项做了概述，现在以具体的实例程序详细解析 SSE 的程序设计过程、技术要点。下面将针对编程准备和编程实例两部分的内容做详细阐述。

1. 编程准备

1）包含的头文件

所有的 SSE 指令函数和__m128 数据类型都在 xmmintrin.h 文件中定义。

```
#include <xmmintrin.h>
```

因为程序中用到的 SSE 处理器指令是由编译器决定的，所以它并没有相关的.lib 库文件。

2）分配数据空间（Data Alignment）

由 SSE 指令处理的每一个浮点数数组必须把其中需要处理的数按每 16 字节（128 位二进制）分为一组，一个静态数组（Static Array）可由__declspec(align(16))关键字声明。

```
__declspec(align(16)) float m_fArray[ARRAY_SIZE];
```

其中 align(16) 表示数据以 16 字节地址对齐，__declspec()是 VC 的关键字。数组名为 m_fArray，类型为 float，大小为 ARRAY_SIZE。

动态数组（Dynamic Array）可由_aligned_malloc 函数为其分配空间。

```
m_fArray = (float*) _aligned_malloc(ARRAY_SIZE * sizeof(float), 16);
```

ARRAY_SIZE * sizeof(float)为_aligned_malloc 函数要申请的空间大小，16 表示对齐方式以 16 字节，该函数返回的指针需要做强制类型转换。

由_aligned_malloc 函数分配空间的动态数组可以由_aligned_free 函数释放其占用的空间。

```
_aligned_free(m_fArray);
```

_aligned_free 函数的参数为待释放的空间指针。

3）__m128 数据类型

该数据类型的变量可用做 SSE 指令的操作数，它们不能被用户指令直接存取，_m128 类型的变量被自动分配为 16 字节。

4）CPU 对 SSE 指令集的支持

如果读者的 CPU 具有 SSE 指令集，读者就可以使用 Visual Studio 2005 提供的对 SSE 指令集支持的 C++函数库，读者可以查看 MSDN 中的一个 Visual C++ CPUID 的例子，它可以帮读者检测其 CPU 是否支持 SSE/SSE 2、MMX 指令集或其他的 CPU 功能。

2. 编程实例

以下实例讲解了 SSE 技术在 Visual Studio 2005 下的应用实例，读者可以在 http://www.codeproject.com/KB/recipes/sseintro/SSE_src.zip 下载示例程序压缩包，同样也需要用户注册并登录后方可下载。该压缩包中含有两个项目，这两个项目是基于微软基本类库（MFC）建立的 Visual C++项目。

1）SSE Test 示例项目

SSE Test 项目是一个基于对话框的应用程序，它用到了 3 个浮点数组参与运算。

```
fResult[i] = sqrt( fSource1[i]*fSource1[i] + fSource2[i]*fSource2[i] ) + 0.5
```

其中 i = 0, 1, 2 ···, ARRAY_SIZE-1、ARRAY_SIZE 被定义为 30 000。数据源数组（Source 数组）通过使用 sin 和 cos 函数给它赋值，用 Kris Jearakul 开发的瀑布状图表控件（Waterfall Chart Control）来显示参与计算的源数组和结果数组。算法处理过程是计算两个数据源的平方根，计算所需的时间（以毫秒“ms”为单位）在对话框中显示出来。现使用 3 种不同的途径来完成计算：纯 C++代码；使用 SSE 指令函数的 C++代码；包含 SSE 汇编指令的代码。

纯 C++代码，代码如下。

```
void CSSETestDlg::ComputeArrayCPlusPlus(
                        float* pArray1,     //[输入]源数组1
                        float* pArray2,     //[输入]源数组2
                        float* pResult,     //[输出]用来存放结果的数组
                        int nSize)          //[输入]数组的大小
{
    int i;
    float* pSource1 = pArray1;               //数据源1
    float* pSource2 = pArray2;               //数据源2
    float* pDest = pResult;                  //结果数据
    for ( i = 0; i < nSize; i++ )            //处理的数目
    {
        //计算两个数据的平方根
        *pDest = (float)sqrt((*pSource1) * (*pSource1) + (*pSource2) * (*pSource2))
        + 0.5f;
        pSource1++;
        pSource2++;
        pDest++;
```

```
    }
}
```

循环处理传入的两个数据的平方根，为了精度，考虑到了浮点数的四舍五入。

下面用具有 SSE 特性的 C++代码重写上面这个函数，使用到的相关指令和函数如表 12-4 所示。

表 12-4 使用的助记符和函数

函数说明	助记符	MMX Intrinsic
将 4 个 32 位浮点数放进一个 128 位的存储单元	movss 和 shufps	_mm_set_ps1
将 4 对 32 位浮点数同时进行相乘操作。这 4 对 32 位浮点数来自两个 128 位的存储单元，再把计算结果（乘积）赋给一个 128 位的存储单元	mulps	_mm_mul_ps
将 4 对 32 位浮点数同时进行相加操作。这 4 对 32 位浮点数来自两个 128 位的存储单元，再把计算结果（相加之和）赋给一个 128 位的存储单元	addps	_mm_add_ps
对一个 128 位存储单元中的 4 个 32 位浮点数同时进行求平方根操作	sqrtps	_mm_sqrt_ps

使用 Visual C++ 2005 的 SSE 指令函数的代码，代码如下。

```
void CSSETestDlg::ComputeArrayCPlusPlusSSE(
                            float* pArray1,     //[输入]源数组1
                            float* pArray2,     //[输入]源数组2
                            float* pResult,     //[输出]用来存放结果的数组
                            int nSize)          //[输入]数组的大小
{
    int nLoop = nSize/ 4;                       //每次处理4个
    __m128 m1, m2, m3, m4;
    __m128* pSrc1 = (__m128*) pArray1;          //强制类型转换
    __m128* pSrc2 = (__m128*) pArray2;          //强制类型转换
    __m128* pDest = (__m128*) pResult;          //强制类型转换
    __m128 m0_5 = _mm_set_ps1(0.5f);
    //m0_5[0, 1, 2, 3] = 0.5，4个32位浮点数赋给128位变量
    for ( int i = 0; i < nLoop; i++ )           //循环处理
    {
      m1 = _mm_mul_ps(*pSrc1, *pSrc1);          //m1 = *pSrc1 * *pSrc1
      m2 = _mm_mul_ps(*pSrc2, *pSrc2);          //m2 = *pSrc2 * *pSrc2
      m3 = _mm_add_ps(m1, m2);                  //m3 = m1 + m2
      m4 = _mm_sqrt_ps(m3);                     //m4 = sqrt(m3)
      *pDest = _mm_add_ps(m4, m0_5);            //*pDest = m4 + 0.5f
      pSrc1++;
      pSrc2++;
      pDest++;
    }
}
```

上述代码实现每次处理 4 对源浮点数的平方根计算。

使用 SSE 汇编指令实现的 C++函数代码，代码如下。

```
void CSSETestDlg::ComputeArrayAssemblySSE(
                            float* pArray1,      //[输入]源数组1
                            float* pArray2,      //[输入]源数组2
                            float* pResult,      //[输出]用来存放结果的数组
                            int nSize)           //[输入]数组的大小
{
    int nLoop = nSize/4;                         //每次处理4个
    float f = 0.5f ;
    _asm
    {
      movss xmm2, f               //xmm2[0] = 0.5
      shufps xmm2, xmm2, 0        //xmm2[1, 2, 3] = xmm2[0]
      mov esi, pArray1            //输入的源数组1的地址送往esi
      mov edx, pArray2            //输入的源数组2的地址送往edx
      mov edi, pResult            //输出结果数组的地址保存在edi
      mov ecx, nLoop              //循环次数送往ecx
start_loop:
      movaps xmm0, [esi]          //xmm0 = [esi]
      mulps xmm0, xmm0            //xmm0 = xmm0 * xmm0
      movaps xmm1, [edx]          //xmm1 = [edx]
      mulps xmm1, xmm1            //xmm1 = xmm1 * xmm1
      addps xmm0, xmm1            //xmm0 = xmm0 + xmm1
      sqrtps xmm0, xmm0           //xmm0 = sqrt(xmm0)
      addps xmm0, xmm2            //xmm0 = xmm1 + xmm2
      movaps [edi], xmm0          //[edi] = xmm0
      add esi, 16                 //esi += 16
      add edx, 16                 //edx += 16
      add edi,  16                //edi += 16
      dec ecx                     //ecx--
      jnz start_loop              //如果不为0则转向start_loop
    }
}
```

上述代码实现每次处理 4 对源浮点数的平方根计算。注意使用嵌入式汇编时，关键字__asm 的使用。

最后，在作者的计算机上运行计算测试的结果如下。

- 纯 C++代码计算所用的时间是 26 毫秒。
- 使用 SSE 的 C++函数计算的时间是 9 毫秒。
- 包含 SSE 汇编指令的时间是 9 毫秒。

以上的时间结果是在 SSE Test 程序的 Release 优化编译后执行程序得出的。

2）SSE Sample 示例项目

SSE Sample 项目是一个基于对话框的应用程序，其中它用下面的浮点数数组进行计算。

```
fResult[i] = sqrt(fSource[i]*2.8); //其中i = 0, 1, 2… ARRAY_SIZE-1
```

这个程序同时计算了数组中的最大值和最小值。ARRAY_SIZE 被定义为 100000，数组

中的计算结果在列表框中显示出来。其中在笔者的电脑上用下面 3 种方法计算所需的时间如下。

- 纯 C++代码计算 6 毫秒。
- 使用 SSE 的 C++函数计算 0 毫秒。
- 使用 SSE 汇编指令计算 0 毫秒。

大家看到，使用 SSE 汇编指令计算的结果会好一些，因为使用了效率增强了的 SSX 寄存器组。但是在通常情况下，使用 SSE 的 C++函数计算会比汇编代码计算的效率高一些，因为 C++编译器的优化后的代码有很高的运算效率，若要使汇编代码比优化后的代码运算效率更高，这通常是很难做到的。

纯 C++代码，代码如下。

```
//输入: m_fInitialArray
//输出: m_fResultArray, m_fMin, m_fMax
void CSSESampleDlg::OnBnClickedButtonCplusplus()
{
  TIME_START;                        //记录算法开始系统时间
  m_fMin = FLT_MAX;                  //float.h中的值，表示最大
  m_fMax = FLT_MIN;                  //float.h中的值，表示最小
  int i;
  for ( i = 0; i < ARRAY_SIZE; i++ ) {                           //循环处理的次数
      m_fResultArray[i] = sqrt(m_fInitialArray[i] * 2.8f); //计算平方根
      if ( m_fResultArray[i] < m_fMin )       m_fMin = m_fResultArray[i];
      if ( m_fResultArray[i] > m_fMax)       m_fMax = m_fResultArray[i];
  }
  TIME_END;                          //记录当前系统时间，显示算法处理时间
  ShowArray(m_fResultArray);  //显示结果数据
}
```

上述程序首先记录算法开始的系统时间，然后对输入的数据源乘以 2.8，接着找出这些数据中的最小值和最大值。随后显示当前处理方法的时间消耗，并显示处理结果。

使用 Visual C++.NET 的 SSE 指令函数的代码，代码如下。

```
//输入: m_fInitialArray
//输出: m_fResultArray, m_fMin, m_fMax
void CSSESampleDlg::OnBnClickedButtonSseC()
{
 TIME_START;                             //记录算法开始系统时间__m128 coeff =
 _mm_set_ps1(2.8f);                      //coeff[0, 1, 2, 3] = 2.8，即打包数据
 __m128 tmp;
 __m128 min128 = _mm_set_ps1(FLT_MAX);   //min128[0, 1, 2, 3] = FLT_MAX
 __m128 max128 = _mm_set_ps1(FLT_MIN);   //max128[0, 1, 2, 3] = FLT_MIN

 __m128* pSource = (__m128*) m_fInitialArray;
 __m128* pDest = (__m128*) m_fResultArray;
 for ( int i = 0; i < ARRAY_SIZE/4;i++ )//每次处理4个
```

```
    {
        tmp = _mm_mul_ps(*pSource, coeff);    //tmp = (*pSource) * coeff
        *pDest = _mm_sqrt_ps(tmp);            //*pDest = sqrt(tmp),计算平方根
        min128 = _mm_min_ps(*pDest, min128); //比较得出较小值
        max128 = _mm_max_ps(*pDest, max128); //比较得出较大值
        pSource++;
        pDest++;
    }
    //从max128、min128找出最小值和最大值
    union u {
        __m128 m;
        float f[4];
    } x;
    x.m = min128;
    m_fMin = min(x.f[0], min(x.f[1], min(x.f[2], x.f[3])));
    x.m = max128;
    m_fMax = max(x.f[0], max(x.f[1], max(x.f[2], x.f[3])));
   TIME_END;                                 //记录当前系统时间，显示算法处理时间
   ShowArray(m_fResultArray);                //显示结果数据
  }
```

在上述程序实现过程中，一个浮点 32 位，每次处理 4 个，共 128 位。在 max128/min128 中继续比较得出最小值和最大值，最后显示处理时间及处理结果。

使用 SSE 汇编指令的 C++函数代码，代码如下。

```
//输入: m_fInitialArray
//输出: m_fResultArray, m_fMin, m_fMax
void CSSESampleDlg::OnBnClickedButtonSseAssembly()
{
  TIME_START;                          //记录算法开始系统时间
  float* pIn = m_fInitialArray;
  float* pOut = m_fResultArray;
  float f = 2.8f;
  float flt_min = FLT_MIN;
  float flt_max = FLT_MAX;
  __m128 min128;
  __m128 max128;
  //使用以下的附加寄存器:xmm2、xmm3、xmm4:
  //xmm2 - 相乘系数
  //xmm3 - 最小值
  //xmm4 - 最大值
  _asm
  {
    movss xmm2, f                      //xmm2[0] = 2.8
    shufps xmm2, xmm2, 0               //xmm2[1, 2, 3] = xmm2[0]
    movss xmm3, flt_max                //xmm3 = FLT_MAX
    shufps xmm3, xmm3, 0               //xmm3[1, 2, 3] = xmm3[0]
```

```
        movss xmm4, flt_min                 //xmm4 = FLT_MIN
        shufps xmm4, xmm4, 0                //xmm3[1, 2, 3] = xmm3[0]
        mov esi, pIn                        //输入数组的地址送往esi
        mov edi, pOut                       //输出数组的地址送往edi
        mov ecx, ARRAY_SIZE/4               //循环计数器初始化
start_loop:
        movaps xmm1, [esi]                  //xmm1 = [esi]
        mulps xmm1, xmm2                    //xmm1 = xmm1 * xmm2
        sqrtps xmm1, xmm1                   //xmm1 = sqrt(xmm1)
        movaps [edi], xmm1                  //[edi] = xmm1
        minps xmm3, xmm1
        maxps xmm4, xmm1
        add esi, 16
        add edi, 16
        dec ecx
        jnz start_loop
        movaps min128, xmm3
        movaps max128, xmm4
    }
    //从max128、min128找出最小值和最大值
     union u {
        __m128 m;
        float f[4];
        } x;
        x.m = min128;
        m_fMin = min(x.f[0], min(x.f[1], min(x.f[2], x.f[3])));
        x.m = max128;
        m_fMax = max(x.f[0], max(x.f[1], max(x.f[2], x.f[3])));
        TIME_END;                           //记录当前系统时间，显示算法处理时间
        ShowArray(m_fResultArray);          //显示结果数据
    }
```

上述实现过程与前一个基本相同，注意嵌入式汇编的关键字__asm。

SSE Test 和 SSE Sample 这两个项目的其他程序请参考光盘 chapter12\SSE 目录。

12.3 使用MMX/SSE 2进行音/视频开发

上面介绍了 MMX/SSE/SSE 2 对数据的一般优化处理过程，体会了使用 SSE 2 指令和 XMM 寄存器的技术过程。现在使用 MMX/SSE 媒体汇编指令进行音频、视频的算法优化和设计。本节包括开发前的技术准备、视频编/解码 MMX/SSE 编程和使用 VC 对 MMX/SSE 2 程序调试。

12.3.1 开发前技术准备

在进行 MMX/SSE 2 开发时由于指令繁多且复杂，开发人员不可能都一一记住，因此汇编指令手册是必不可少的。表 12-5 列出了 Intel 网站提供的常用文档，包括基本结构、指令集 A～Z、系统编程指南和优化参考手册。

表 12-5 Intel 汇编指令参考手册

文档标题	序号
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture （基本结构）	253665.pdf
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M （指令 A～M）	253666.pdf
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z （指令 N～Z）	253667.pdf
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide. （系统编程指南）	253668.pdf
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide （系统编程指南）	253669.pdf
Intel® 64 and IA-32 Architectures Optimization Reference Manual（优化参考）	248966.pdf

其中的 253666.pdf、253667.pdf 是要经常查用的，这些是 Intel 的最新指令参考手册，包括了 MMX/SSE 2/SSE 3 等指令集的详细信息：指令名全称、指令描述、指令操作过程、指令影响的标志位、对应 Intrinsic 指令等。

使用 MMX/SSE 2 开发多媒体程序，一方面要对指令有一定的熟悉程度，熟练掌握常用的指令功能及操作；另一方面要对算法彻底理解，以便于根据指令特点重构、改写算法。开源的音/视频编/解码算法工程几乎都提供了核心函数的汇编优化，一方面技术人员可以直接使用这些优化汇编，另一方面模仿这些汇编优化的结构、指令编排特点等，编写出自己的高效汇编程序。

另外，在优化算法时，C/C++与汇编的混合编程是必需的。如汇编优化算法在一个*.asm 文件中，主程序框架是 C/C++编写，则大家在编写汇编时就需要时刻注意寄存器的现场保持、函数返回值、参数传递与获取，汇编函数命名需要加“_”等。在 C/C++中使用嵌入式汇编，也更应注意寄存器等的使用规则，以下是嵌入式汇编编程时特别要遵循的几个原则。

1）一般原则

- 自由使用通用寄存器：EAX、EBX、ECX 和 EDX。
- 若要使用其他寄存器，则先压堆栈保存，使用完后需恢复，例如：

```
__asm{
 push ebp                    //ebp压入堆栈
 push esp                    //esp压入堆栈
```

```
    ……….//使用ebp和esp
    pop esp                    //ebp弹出堆栈
    pop ebp                    //esp弹出堆栈
}
```

2）__asm 单独使用

```
 __asm mov eax,val0       ; eax = val0
__asm mov ebx,val1        ; ebx = val1
```

3）汇编不区分大小写

```
 __asm{
    mov  eax,ecx
    MOV EAX,ECX                //与上句相同
}
```

4）返回值直接用 eax

```
 int my_fun(……/*parameters*/)
{
    int i_ret;                 //函数返回值
    ……                         //函数其他语句
    __asm mov eax,i_ret        //代替return i_ret；但编译器会发出警告，但结果正确
}
```

5）嵌入式汇编

嵌入式汇编使用__asm 或_asm 而不使用 asm，微软的 MSDN 给出了建议参考。

12.3.2 MMX/SSE 2视频编/解码编程

视频编/解码算法处理的数据是海量的图像数据。底层模块是对图像块 16×16、8×8 等的规则处理，如块的 DCT 变换与反变换 IDCT、量化与反量化、运动补偿插值、计算 SAD 值等，这些算法非常适合数据打包并行处理、汇编编程优化。不同的平台的计算瓶颈不同，如 PC 的 CPU 平台，图像数据放置在高速的 DDR2 内存；而 DSP 如 DM642 平台，图像数据则放置在低速的片外 SDRAM。普通 CPU 流水线不明显，DM642 CPU 需要调整程序代码使之能并行流水。这里介绍使用 MMX/SSE 2 进行视频编/解码编程优化，第 13 章将介绍 DM642 实现视频编/解码。

CPU 汇编指令 MMX/SSE 2 设计的初衷就是提高图形、数字音/视频处理的效率。其中一个主要的特点就是数据打包，一条指令处理多个数据。MMX 使用的寄存器是 64 位的，SSE 2 使用的寄存器是 128 位的。这样 8 比特的像素点就可以一次处理多个像素。另外数据的吞吐率也提高了，一次读写不再是一个像素点，而是 8/16 个像素点。视频编/解码算法主要执行加、减、乘、除操作。加/减/乘有专门的指令支持，而对于除法运算 MMX/SSE 2 则没有专门的指令支持，可以通过移位或查表来实现。

视频编/解码算法 MPEG-4/H.264 等都是基于块的混合编码技术，工程实现 XviD 和 X264 提供了部分核心函数的汇编优化，包括各种 CPU 汇编版本：MMX、MMX-Ext、SSE、3DNow!、SSE 2、IA64asm、Power PC（PPC）等。

1. MMX 编程优化视频编/解码算法

MMX 寄存器 MM0～MM7 是 64 位的，可以作为 8 字节、4 个字、2 个双字来使用。在视频算法中，像素点 YUV 的精度是 8 比特，DCT 数据类型是双字节 short 短整型。这样就能够把图像 8×8 块的一行 8 个像素点即 8 字节放置在一个 64 位的 MMX 寄存器中，。然后使用数据打包的运算指令完成加/减操作。

前面已经展示了演示项目 MMX8/MMX32，展示了在 C++环境中使用 Intrinsic 指令（或函数）和嵌入式汇编__asm{…}的方法来实现 MMX 优化算法的过程。这里展示编写优化汇编文件*.asm，然后汇编中的函数被 C 语言调用的 MMX 技术来优化视频编码算法的过程。

视频编/解码算法的核心函数包括 I 帧编码的图像 8 位像素减并扩展为 16 位、DCT 变换、系数量化、IDCT 变换、系数反量化、16 位嵌位成 8 位并加回源图像的块重建、预测、zigzag 扫描和熵编码。P 帧编码的运动估计、计算 SAD 值、图像块滤波插值、运动补偿 MC 插值、残差的 DCT/IDCT 变换、量化/反量化、重建加回源图像、zigzag 扫描和熵编码。所有处理模块都是基于 16×16 的宏块即 4 个 8×8 的亮度块和两个 8×8 的色度块的处理，所以说视频算法非常适合 MMX 汇编优化。

下面通过一个实例来分析 MMX 编程优化的过程。在块系数编码时，如果所有系数均为零，则不编码而跳过。这个标识即 CBP 块编码模式。6 个 8×8 块用 6 比特分别来标识是否被编码。CBP 的 6 位从高到低分表标识亮度块 0、1、2、3 和色度块 4、5。

1）C 语言算法

```
uint32_t calc_cbp_plain(const int16_t codes[6 * 64])
{
 int i, j;
 uint32_t cbp = 0;
 for (i = 0; i < 6; i++) {
        for (j=1; j<64;j++) {                          //只扫描AC系数，j=1
               if (codes[64*i+j]) {
                              cbp |= 1 << (5-i); //块有一个非零系数，则块标识为需要编码
                      break;
               }
        }
 }
 return cbp;
}
```

C 语言算法使用了有一个非零终止循环的 break 语句，并且每次读取一个系数。

2）MMX 优化

```
BITS 32                         ;以下是32位代码
;=====================================================================
; Macros
;=====================================================================
%macro cglobal 1
%ifdef PREFIX      ;实际有效，汇编定义的函数前面需要加_
 global _%1        ;C语言才能调用该函数
```

```
    %define %1 _%1
 %else
    global %1
 %endif
%endmacro
;=============================================================================
; Local data
;=============================================================================
%ifdef FORMAT_COFF
SECTION .rodata data
%else
SECTION .rodata data align=16  ;实际有效，只读数据段，16字节对齐
%endif
ALIGN 16
ignore_dc:
 dw 0, -1, -1, -1, -1, -1, -1, -1  ;定义的常量数组，名称ignore_dc，0是LSB
;=============================================================================
; Code
;=============================================================================
SECTION .text
cglobal calc_cbp_mmx
;-----------------------------------------------------------------------------
; uint32_t calc_cbp_mmx(const int16_t coeff[6][64]);
;-----------------------------------------------------------------------------
ALIGN 16                      ;以下地址16字节对齐
calc_cbp_mmx:
  push ebx                    ;压栈ebx寄存器
  push esi                    ;压栈esi寄存器
  mov esi, [esp + 8 + 4]      ;获取coeff地址
  xor eax, eax                ;eax存储cbp,cbp = 0
  mov edx, (1 << 5)           ;edx=1<<5
  movq mm7, [ignore_dc]       ;mm7 = 0xffff,ffff,ffff,0000
.loop
  movq mm0, [esi]             ;mm0=a3,a2,a1,a0
  movq mm1, [esi+8]           ;mm1=a7,a6,a5,a4
  pand mm0, mm7               ;mm0=a3,a2,a1,0 , 即把直流DC清零
  por mm0, [esi+16]           ;以下把一个块的另外7行数据读取并做或运算
  por mm1, [esi+24]
  por mm0, [esi+32]
  por mm1, [esi+40]
  por mm0, [esi+48]
  por mm1, [esi+56]
  por mm0, [esi+64]
  por mm1, [esi+72]
  por mm0, [esi+80]
  por mm1, [esi+88]
```

```
  por mm0, [esi+96]
  por mm1, [esi+104]
  por mm0, [esi+112]
  por mm1, [esi+120]
  por mm0, mm1
  movq mm1, mm0              ;传输64位整数
  psrlq mm1, 32              ;4字（64位）逻辑右移32位
  lea esi, [esi + 128]       ;esi指向下一个8×8块
  por mm0, mm1
  movd ebx, mm0
  test ebx, ebx              ;test指令相当与and指令，但不改变操作数，只改变标志位
  jz .next                   ;若为零，则跳转到.next，否则记录下cbp
  or eax, edx                ;cbp |= 1 << (5-i)
.next
  shr edx,1                  ;逻辑右移
  jnc .loop                  ;无进位时转移
  pop esi                    ;esi弹栈
  pop ebx                    ;ebx弹栈
  ret
```

在上述优化中，对块的所有数据都判断，一次读取 64 比特 4 个处理单元，循环一次对块的 64 个数据都做处理，循环 6 次处理 6 个块。

2. SSE/SSE 2 编程优化视频编/解码算法

SSE/SSE 2 汇编是 MMX 的扩展，增加了 8 个 128 位的寄存器 XMM0～XMM7，两个 4 字打包。对于 8 比特像素点，XMM 寄存器可存储 16 个像素点，8 个 short 型 DCT 系数。对于 DCT/IDCT 和 QUANT/IQUANT 等短整型数据算法非常适合，数据吞吐量比 MMX 指令的寄存器高一倍。SSE 优化的算法比 MMX 优化的算法性能高很多。

```
BITS 32                              ;以下是32位代码
;=============================================================================
; Macros
;=============================================================================
%macro cglobal 1
 %ifdef PREFIX                       ;实际有效，汇编定义的函数前面需要加_
     global _%1                      ;C语言才能调用该函数
     %define %1 _%1
 %else
     global %1
 %endif
%endmacro

%macro LOOP_SSE2 1                       ;处理一个8×8块
 movdqa xmm0, [edx+(%1)*128]             ;%1为传入的参数，读取128位
 pand xmm0, xmm7                         ;把直流分量DC清零
 movdqa xmm1, [edx+(%1)*128+16] ;读取8×8块的下一行
```

```
  por xmm0, [edx+(%1)*128+32]      ;读取128位并做或运算
  por xmm1, [edx+(%1)*128+48]
  por xmm0, [edx+(%1)*128+64]
  por xmm1, [edx+(%1)*128+80]
  por xmm0, [edx+(%1)*128+96]
  por xmm1, [edx+(%1)*128+112]
  por xmm0, xmm1                   ;xmm0 = xmm1 = 128 bits信息
  psadbw xmm0, xmm6                ;xmm0包含2个双字的和
  movhlps xmm1, xmm0               ;把双字的高位移动到地位
  por xmm0, xmm1                   ;组合
  movd ecx, xmm0                   ;ecx置位，则找到非零值
  test ecx, ecx
%endmacro
;===========================================================================
; Data (Read Only)
;===========================================================================
%ifdef FORMAT_COFF
SECTION .rodata data
%else
SECTION .rodata data align=16      ;常量段，只读数据段，16字节对齐
%endif
ALIGN 16                           ;地址16字节对齐
ignore_dc:                         ;定义字，名称为ignore_dc
 dw 0, -1, -1, -1, -1, -1, -1, -1
;===========================================================================
; Code
;===========================================================================
SECTION .text
;---------------------------------------------------------------------------
; uint32_t calc_cbp_sse2(const int16_t coeff[6*64]);
;---------------------------------------------------------------------------
ALIGN 16
cglobal calc_cbp_sse2
calc_cbp_sse2:
 mov edx, [esp+4]                  ;coeff[]
 xor eax, eax                      ;cbp = 0
 movdqu xmm7, [ignore_dc]     ;屏蔽dc值
 pxor xmm6, xmm6              ;xmm6=0
 LOOP_SSE2 0                  ;处理第一个块
 test ecx, ecx                ;测试ecx
 jz .blk2                     ;等于零跳转到下一个亮度块
 or eax, (1<<5)               ;否则记录当前块的CBP
.blk2
 LOOP_SSE2 1                  ;处理第二个块
 test ecx, ecx                ;测试ecx
 jz .blk3                     ;等于零跳转到下一个亮度块
```

```
    or eax, (1<<4)              ;否则记录当前块的CBP
  .blk3
    LOOP_SSE2 2                 ;处理第三个块
    test ecx, ecx               ;测试ecx
    jz .blk4                    ;等于零跳转到下一个亮度块
    or eax, (1<<3)              ;否则记录当前块的CBP
  .blk4
    LOOP_SSE2 3                 ;处理第四个块
    test ecx, ecx               ;测试ecx
    jz .blk5                    ;等于零跳转到色度块
    or eax, (1<<2)              ;否则记录当前块的CBP
  .blk5
    LOOP_SSE2 4                 ;处理第五个块
    test ecx, ecx               ;测试ecx
    jz .blk6                    ;等于零跳转到色度块
    or eax, (1<<1)              ;否则记录当前块的CBP
  .blk6
    LOOP_SSE2 5                 ;处理第六个块
    test ecx, ecx               ;测试ecx
    jz .finished                ;等于零则结束
    or eax, (1<<0)              ;否则记录当前块的CBP
  .finished
   ret                          ;函数返回，eax包含最后的结果CBP
```

在上述优化中，一次读取 8×8 块的一行 DCT 系数数据，利用了与 0 作差的绝对值的和完成加运算。

其他算法的 MMX/SSE 2 汇编指令优化与上述算法的优化大同小异。主要思想是数据的打包处理，利用 64 位、128 位 CPU 寄存器代替使用存储器，既提高运算速度，又提高数据存取速度。

12.3.3 使用VC调试MMX/SSE 2程序

不同的 CPU 其汇编指令内容不同，则调试方法也就不同：C/C++环境的 Intrinsic 指令编程、嵌入式汇编编程同 C/C++普通调试方法相同；C/C++和汇编文件的混合编程调试时则同普通调试方法不同。前述的 MMX8/MMX32/SSE Sample 均是 C/C++环境下的编程与调试，调试方法没有特别之处，只是在观察变量时，需要开启观察 CPU 寄存器、MMX/SSE 2 的寄存器。

1. 嵌入式汇编与 Intrinsic 指令汇编

这两种方法的汇编编程优化算法在调试汇编程序时，在源文件中直接设置断点，然后运行程序。程序运行到断点处中止，断点设置如图 12-16 所示。然后运行 VC++ 2005 菜单中的“调试”→“窗口”→“寄存器”命令，启动“寄存器”对话框，如图 12-17 所示。按快捷键“F10”单步运行即可发现寄存器的值发生变化。

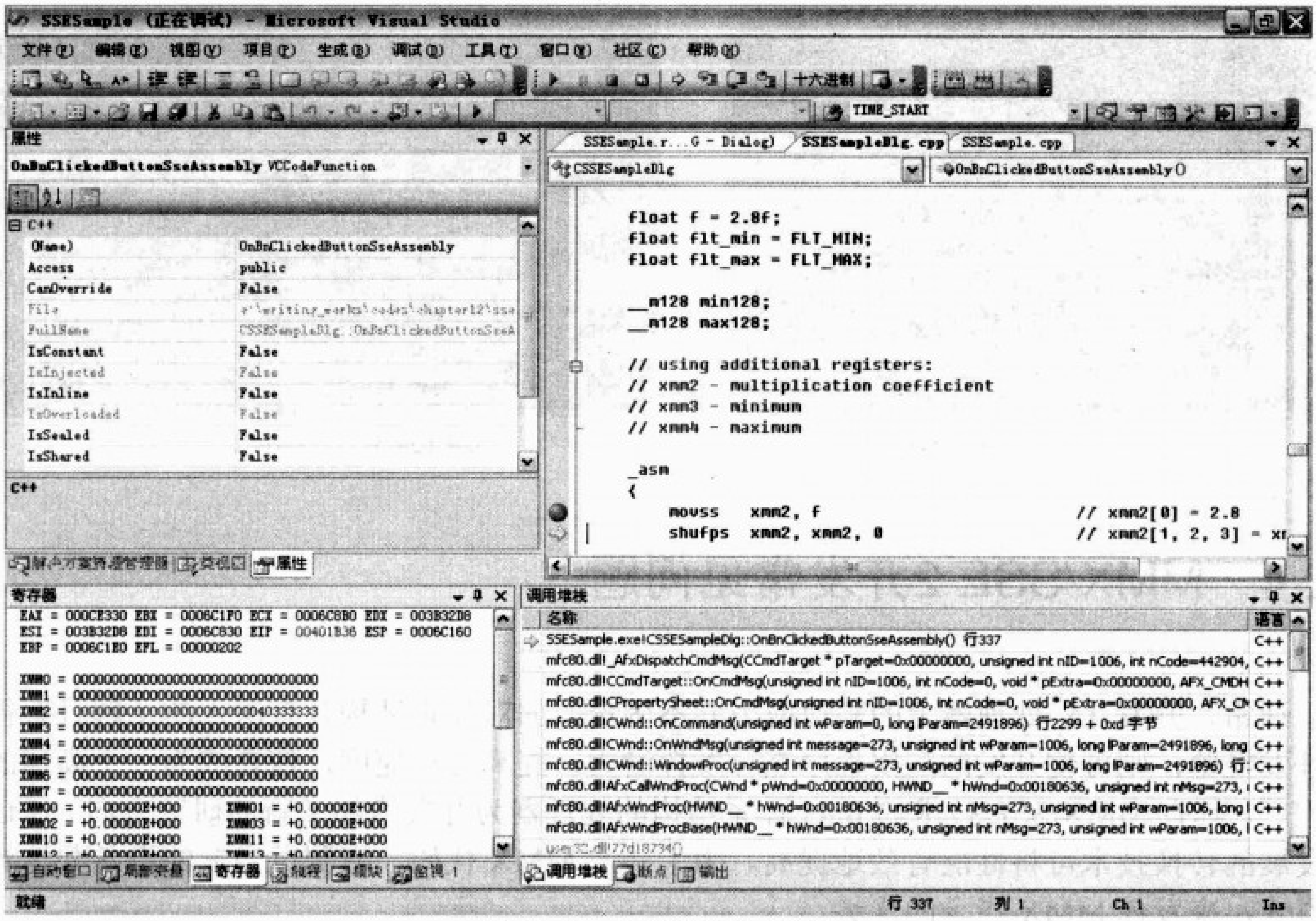


图 12-16 断点设置、调试

图 12-17 “寄存器”对话框

2. C/C++与汇编混合编程

在这种混合编程调试时，需要用到 VC++的反汇编窗口。首先在 C/C++的汇编函数调用处设置断点，然后运行程序到断点处，运行 VC++ 2005 菜单中的“调试”→“窗口”→“反汇编”命令，启动“反汇编”对话框，如图 12-18 所示。

然后按“F11”键逐句运行，跳入到汇编函数入口，汇编调试如图 12-19 所示。

VC++ 2005 开发的程序有两个版本：Debug 和 Release。两者的运行效率差别极大，所以在多次编译和调试后发布最终版时，务必使用 Release 发布程序。

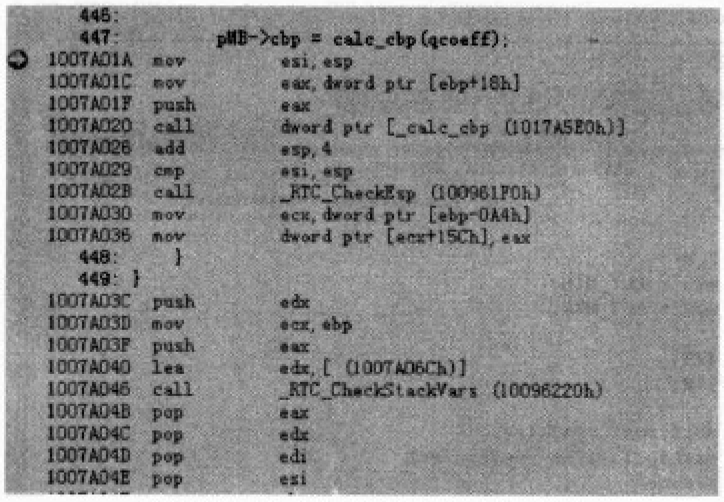


图 12-18 “反汇编”对话框

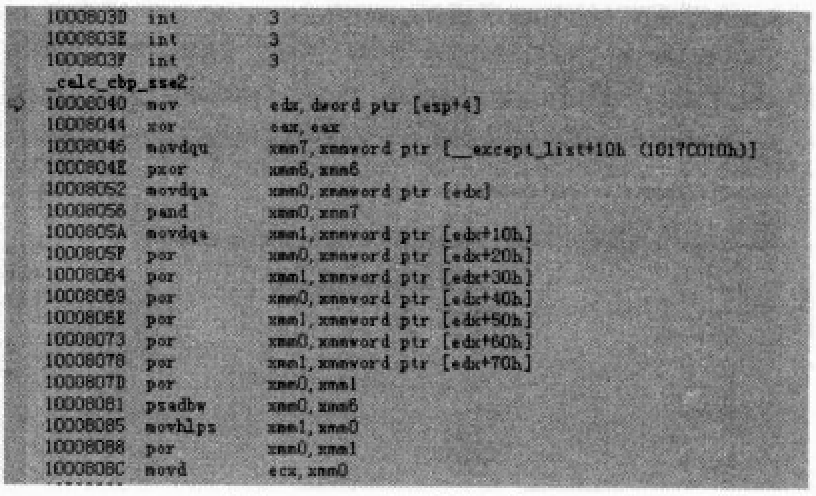


图 12-19 汇编调试

12.4 MMX/SSE 2开发常见问题

通常，开发 IA 处理器上的高速应用并不困难。如对体系结构具有深入的了解且有很好的开发经验，则可使开发的应用程序充分发挥潜力，提高运行速度，使之成为一个高速的应用程序。具有 MMX/SSE 技术的 Intel 体系结构的处理器为开发增加了新的研究方向。通过不断发展的转换技术可将性能有效地提高。为了开发 MMX 代码，应测试当前的设备并选择最佳方式，来发挥 MMX 指令的优势。

12.4.1 制订计划

无论是修改现有应用程序还是创建新的应用程序，在使用 MMX 指令来达到最佳性能时，需要考虑几个方面的因素。通常要先找到计算量集中的代码段，且该代码段应是可修改为整数实现的，并有效支持高速缓存的体系结构。Intel 性能工具箱提供了几个工具来辅助进行判断和调整。

在进行这些工作前，应先回答下列问题。

- 哪部分代码可用 MMX 技术改进。
- 使用 MMX 技术是否可使当前算法达到最佳。
- 代码是整型的还是浮点型的。
- 如何安排数据。
- 代码是 8 位、16 位还是 32 位的。
- 应用程序是否需同时在有 MMX 技术和无 MMX 技术的处理器上运行，是否可用 CPUID 来创建一个可自适应的实现。

下面依次介绍以上几个问题。

12.4.2 哪部分代码可使用MMX技术改进

大多数应用程序都有几个计算高度集中的代码段。例如，语音、音频压缩算法和滤波器、视频显示例程和绘制着色例程。这些例程通常较小，有重复的循环，对 8 位或 16 位整数操作，且占用了应用程序相当大的处理时间。这些例程使用 MMX 技术优化库代码进行转换可获得最大的性能提高。将这些循环封装到优化的 MMX 技术库，无论是否具有 MMX 技术的平台，都将更灵活地给予支持。性能优化工具，比如 Intel 的 VTune 可视化调试工具，可用于把计算量最大的核心模块的代码分离出来。一旦将这些代码找到，将判断当前算法和修改后的代码哪一个具有最佳性能。在某些情况下，可能会改变算法中某些类型的操作来提高性能，用 MMX 指令性能来匹配算法是达到最佳性能的关键。

12.4.3 代码是浮点型还是整型

在明确需要使用 MMX 优化的模块后，紧接着判断算法包含的是浮点型数据还是整型数据。

如果当前算法是用整型数据实现的，则可简单判断一下算法中占用大多数 CPU 时钟周期的代码部分在哪里。一旦判断出来，即用 MMX 指令重新实现这些部分。

如果算法中存在浮点型数据，则判断一下使用浮点型数据的原因。使用浮点操作通常有以下几个原因：性能、范围和精确度。如果性能因素是算法使用浮点数的原因，那么可考虑用 MMX 整型代码替换算术来提高性能。

如果数据范围或精确度是算法使用浮点数进一步适应可能的范围或精确度的要求，那么可考虑是否能用整型数据达到所要求的范围和精确度，如果不能，则代码最好仍用浮点代码。

如果使用 MMX 代码，需要牢记的一点是 8 个 MMX 寄存器是浮点寄存器的别名。由 MMX 指令切换为浮点指令将增加多达 50 个时钟周期，因此最好将这种切换次数减少到最小。不要将 MMX 代码和浮点代码在指令级进行内部混用，如果一个应用程序频繁地进行 MMX 代码和浮点代码间的切换，则可考虑将其修改为只用 MMX 指令流或浮点指令流来最大限度地降低由切换带来的额外迟延。

当需混用浮点指令或 MMX 指令编写应用程序时，下列准则可用来分离指令执行。

- 将 MMX 指令流和浮点指令流分为只包含单一类型代码的指令流。
- 在转换中，寄存器内容无效。
- 在 MMX 代码结束时，使用 EMMS 指令置浮点标志字为空。
- 在浮点代码结束时，将堆栈置空。

例如：

```
FP_Code:
... /*离开时，将浮点栈置空*/
MMX_Code:
EMMS /*MMX寄存器空*/
FP_Code1:
... /*离开时，将浮点栈置空*/
```

有关浮点编程模型的其他内容，请参见《奔腾处理器系列开发者手册》第三卷，体系结构和编程（序号 241430）。

12.4.4 EMMS准则

如果程序中有浮点类型的数据需要处理，则在 MMX 代码结束时必须使用 EMMS 指令。

由于 MMX 寄存器是浮点寄存器的别名，故一定要在使用浮点指令前清空 MMX 寄存器。使用 EMMS 指令清空 MMX 寄存器并置浮点标志字（TW）的值为空（即为全 1）。该指令应在每个 MMX 代码段的结束时插入，以避免执行浮点代码时产生浮点栈的溢出错误。

12.4.5 CPUID的检测MMX技术的用法

在编程 MMX/SSE 等汇编指令前，需要判断当前 CPU 所支持的 MMX 技术。

可用两种方法在应用程序中使用 MMX 技术：第一种方法是在应用程序安装时检测是否具有 MMX 技术，如果具有 MMX 技术，则可安装相应的库代码；第二种方法是在程序执行时检测，并于运行时装入合适的库代码，这对于网络上的程序非常有效。

为判断读者的处理器是否具有 MMX 技术，应用程序应检测 Intel 体系结构的特性标志。CPUID 指令将特性标志返回到 EDX 寄存器中，程序可基于这一结果为系统选择合适的执行代码。特征标志字的第 23 位指示是否具有 MMX 技术。当该位为 1 时，表示具有 MMX 技术。下列代码段从特性标志字中提取值到 EDX 中，并检测是否具有 MMX 技术。有关 CPUID 用法的其他内容请详见《用 CPUID 指令识别 Intel 处理器》应用程序部分 AP-485，（序号 241618）。

```
... ;CPUID 指令识别是否具有MMX
... ;识别Intel处理器
MOV EAX, 1 ;调用申请特性标志
CPUID ;0FH,A2H,CPUID 指令
test EDX, 00800000H ;在特性标志位(23 位)=1 时为MMX技术存在
JNZ Found
```

12.4.6 数据对齐

很多编译器允许通过控制来指定对齐变量，通常可使变量按合适的边界对齐。但是，如果读者发现有些变量未按指定边界对齐，则请使用下列 C 语言写的算法对齐地址空间。这是按 64 位边界来对齐 64 位变量，一旦对齐后，对该变量的每次访问将节省 3 个时钟周期。

```
if(NULL==(new_ptr=malloc(new_value+1)*sizeof(var_struct)) //申请空间
mem_tmp=new_ptr;
mem_tmp/=8;                                                //地址整除8
new_tmp_ptr=(var_struct*)((Mem_tmp+1)*8);                  //修正，以整除8为目标
```

另一种改进数据对齐的方法是将数据复制到已按 64 位边界对齐的地址中，当数据访问

频繁时，可有效地提高性能。

在通常情况下，编译器将非静态的数据都分配到堆栈，而且通过存储于堆栈的 64 位数据很方便地实现。如果可能，一定要使堆栈对齐，下列代码是使堆栈对齐的起始函数和结束函数。

```
Prologue:
push ebp              ;保存IH 的帧指针
mov ebp, sep          ;生成新的帧指针
sub ebp, 4            ;建立栈指针的空iBq
and ebp, 0FFFFFFC     ;按64位对齐
mov [ebp], esp        ;保存旧的栈指针
mov esp, ebp          ;复制对齐的指针
sub esp, FRAMESIZE    ;分配空间
... callee saves, etc
epilogue:
... callee restores, etc
mov esp, [ebp]
pop ebp
ret
```

对那些频繁访问而又无法对齐的数据，可将它们复制到已对齐的存储空间去。

12.4.7 数据安排

MMX 技术利用 SIMD 技术来开发大多数多媒体算法中固有的并行机制。为了利用 MMX 代码来提高性能，内存中的数据应按下列准则进行格式化。

请考虑一个将 16 位偏置数据加到一个向量的全部 16 位元素上的简单例子。在一般标量代码中，在循环开始时将偏置数据读入一个寄存器。在另一个寄存器中访问向量元素，并且一次只完成一条加法。

将例程改换成 MMX 代码后，由于 MMX 指令可通过 MOVQ 指令一次处理 4 个向量元素，并通过 PADDW 指令一次完成 4 个加法。将使速度提高 4 倍。但是为了达到这一速度，在做加法时需要将偏置数据复制 4 份放在 MMX 寄存器中。

在原始的标量代码中，只有一个偏置数据位于内存中。用 MMX 指令可通过不同方法将 4 个偏置数据的备份放入 MMX 寄存器中，或者通过预先对内存格式化，复制 4 个偏置数据。然后，只需在循环开始时，用 MOVQ 指令读取这些备份数据，就可使速度提高 4 倍。

由 MMX 技术定义的新型 64 位成组数据类型可通过对未对齐的数据进行优化而发挥更大的能力。在使用 MMX 指令和其他成组数据类型时，有很多算法。因为有的数据访问模式是未对齐的，如 FIR 过滤器这样一个简单的例子。FIR 按分支系数的数量长度来产生有效的向量过滤器。如果过滤器对数据元素 i 的操作是一个从数据元素 j 开始产生的向量点（data[j]*coeff[0]+data[j+1]*coeff[1]+...+data[j+num_of_taps-1]*coeff[num_of_taps-1]），那么过滤器对数据元素 i+1 的操作开始于数据元素 j+1。

注意：

复制和填充技术解决了未对齐问题，但要避免由于增大数量而造成未对齐数据访问的额外开销增大，读者在开发代码时，应考虑这种交换并通过选择开关来达到最佳性能。

12.4.8 应用程序最后的调整

在应用程序的功能正确后，最佳的调整方法是使用一个性能分析器来检测在系统中运行的应用程序。Intel 的 VTune 是一个可视化的应用程序性能剖析工具，它能够帮助开发人员判断其应用程序中修改何处的代码可提高性能，算法处理的流程以不同的颜色标注。

12.5 本章小结

海量的音/视频数据、图形、网络流媒体等对 CPU 的媒体处理能力要求越来越高，从 Intel 最早的 MMX 专门的媒体处理、多媒体扩展指令到后来的 SSE/SSE 2/SSE 3/SSE 4，指令越来越丰富，处理能力越来越强。核心算法的 MMX/SSE 2 汇编文件优化，算法系统采用 C/C++ 环境构建框架。使用 Intrinsic 汇编指令、嵌入式汇编和 C/C++与汇编文件混合编程的方式实现核心算法的汇编优化。CPU 寄存器从最早的 MMX 的 64 位到现在的 SSE 2 的 128 位，数据吞吐能力提高了几倍，打包的数据一次完成多个处理：SIMD 单指令多数据处理。

第13章

用DM642实现视频编/解码技术

在视频编/解码的硬件开发中，ASIC专用芯片方案开发投入少、研发周期短，早期的低成本、高利润的音/视频编/解码硬件产品使商家盈利颇丰而受到不少追捧。但是ASIC方案的产品无法升级，开发用户对芯片内部程序无法进行修改。可编程、高性能DSP几乎已经替代了ASIC专用芯片方案，DSP系统开发用户完全掌握知识产权、可升级、易维护。

TMS320DM642DSP（简称DM642）是美国德州仪器公司（TI）于2003年推出的一款针对数字媒体处理的可编程DSP芯片，现已经在数字音/视频处理中被大量的应用，如视频监控行业、数字图像算法处理等。芯片支持消费类和工业级两种应用。DM642针对音/视频处理集成了必需的外围设备，如视频端口VPort、音频端口、网络端口和PCI外设等，为构建音/视频处理系统节省了大量的外设，成本也大大降低。并且已经被证明DM642相对于其他类型ADI的Black Fin系列、Philips的Trimedia PNX系列等，是一款优秀的数字媒体处理方案。特别是多路视频监控、高端应用等，其他DSP都是不可比拟的。另外TI DSP开发文档、应用笔记相当丰富，每一种典型应用TI几乎都有案例供参考，并且C编译器效率非常高，编排优秀的C代码可以达到汇编语言的90%。

本章主要介绍使用DM642开发视频编/解码技术的过程和算法优化过程。

本章重点

- 数字媒体处理器TMS320DM642DSP概述
- DSP软件开发环境CCS
- 用C语言进行视频算法框架编程
- DM642实现视频算法优化
- DM642优化视频算法常见问题

13.1 数字媒体处理器TMS320DM642DSP概述

TMS320DM642 是基于 C6000DSP 家族的 C64x CPU 核，主要是针对数字媒体处理领域应用的一款高性能媒体处理器，它在 C64x 核的基础上整合了众多的外围设备与接口于一体，有 4 个视频口，均可输入或输出，支持 BT656 视频格式、100MB 以太网口、多路音频串口、66MHz PCI 等，可支持 4 路 D1 图像，30 帧/秒的 MPEG-2 的实时压缩。TI 的数字媒体处理器不仅能够以 Microsoft WMV 9 格式提供高清晰视频流，而且还能够提供 MPEG-4、MPEG-2 及 MPEG-1。此外，该器件还支持 H.264。DM642 提供 500MB/600MB/720MB 主频的 DSP，DM642 芯片的功能如图 13-1 所示。

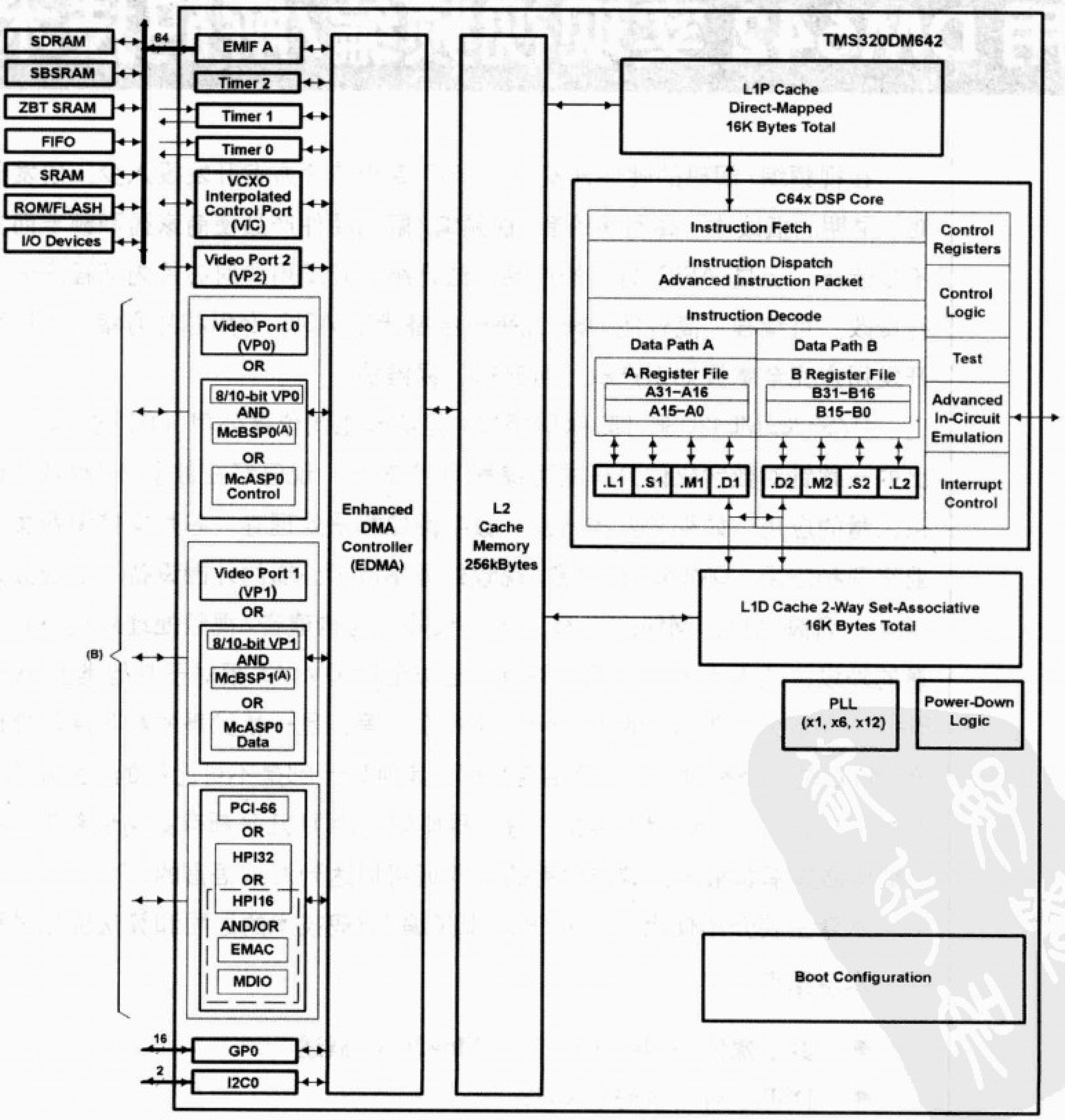


图 13-1 DM642 芯片功能图

1．DM642 的 CPU 单元

基于 C64x CPU 的 DM642，为了满足视频和图像实时处理的需要，DSP 采用 Veloci TI 体系结构和高级超长指令字结构，使得在一个指令周期内能够并行处理多条指令。C64x 的 CPU 可并行 8 条指令，每条指令都是条件指令，600MHz 的 CPU 最高可达 4800MIPS，相当于 4.8GB 的 Intel CPU。具有两个通用寄存器组（A 和 B，各 32 个 32bit 通用寄存器），8 个功能单元（L1、L2、S1、S2、M1、M2、D1、D2）可同时处理数据。两个通道（LD1 和 LD2）从内存读取数据，两个通道（ST1 和 ST2）写数据到内存，两个数据地址通道（DA1 和 DA2）产生访问数据的地址，两个寄存器组数据交叉通道（1X 和 2X）供交叉访问 A、B 两端的数据。

2．DM642 Cache 结构

DM642 的 CPU 与一级程序高速缓存 L1P 和一级数据高速缓存 L1D 直连，两块 Cache 的大小均为 16KB，工作在 CPU 全速访问状态，用户不能进行直接访问。二级缓存 L2 有 256KB，二级缓存的分段和大小分配也有很多变化。部分为 Cache，部分为 SRAM，用户可以直接访问 SRAM，Cache 部分只能通过 Cache 控制命令访问。L2 一般作为 EDMA 的缓冲区缓冲视频图像，L2 是非常宝贵的资源，需要高效使用。CPU 访问数据的过程是首选 CPU 访问 L1P/L1D，如果没有命中则访问 L2，如果还没有访问到要求的数据，则最后访问外部的 SDRAM。

3．DM642 增强 DMA 控制器（EDMA）

DM642 的 EDMA 控制器是一个高效的数据传输设备，在每个 EDMA 周期内可以处理 8 比特，在 CPU 主频是 600MHz 的情况下（EDMA 的频率是 CPU 的频率的 1/2），每秒可以传输 2.4GB。EDMA 执行所有的 L2、外部存储器、外设之间的数据搬移。这些数据包括 CPU 发起、事件触发的传输、主机设备访问、Cache 的维护和非 Cache 的存储器访问等。C64x EDMA 包含 64 个 EDMA 通道，它们服务于外设和外部存储器。对于 DM642，每个通道都有一个特殊的事件与之对应。这些特殊的事件被 EDMA 事件寄存器捕捉。每个事件的优先级都被单独地指定在相应的 EDMA 参数 RAM 中。

4．DM642 的视频端口

视频接口外设能够运行在 3 种方式下，提供视频捕获、视频预览和传输流接口模式。

1）视频捕获模式

- 捕获速率到达 80MHz。
- 支持 2 路 8bit、10bit 数字视频输入，数字视频为 YUV422 格式。
- 支持 1 路 16/20bit 数字视频输入，为 YUV422 格式，可配置为 2 路 8-bit 视频输入。
- 支持 YUV422 到 YUV420 水平转换，以及 8bit YUV422 模式下的亚采样。
- 与视频 AD/DA 芯片无缝连接。

2）视频显示模式

- 显示速率能达到 110MHz。

- 2 路连续视频输出，数字视频输出为 YUV422 格式，8/10bit 精度。
- 1 路连续 Y/C16/20bit 数字视频输出，YUV422 格式，可配置为 2 路。
- YUV420 到 YUV422 格式转换，在 8-bitYUV422 模式下，输出 2 倍插值。
- 能产生行同步、场同步信号和消隐信号。

3）传输流接口（TSI）捕获模式

- 传输流接口能以 8bit 并行，最大 30Mbps 的速率接受数据。
- 传输流接口捕获模式捕获 MPEG-2 传输数据，在传输流中将音频、视频和数据程序流全部复合成一个传输流。MPEG-4 传输流使用定长的数据包，在分配信道容量给音频、视频及数据服务方面有较高的灵活性。
- 支持同步检测，数据捕获在时钟上升沿来临的时候并行数据接收。
- 使用硬件计数器机制为捕获的数据包打时间戳。

5. 以太网口

10Mbps 或 100Mbps 模式自适应，能工作在半双工或全双工模式下，具有硬件流控制及服务质量保证（QOS）支持。

6. 多路音频串口（McASP）

DM642 的多路音频串口功能是为了多路音频应用，而对通用音频串口进行了优化。

McASP 使用相当灵活，与 ADC、DAC 器件、数字音频接口接收器（DIR）等无缝直连。

7. DM642 的其他外设

- 内存扩展接口 EMIF：DM642 的 EMIF 是 64bit，和同步内存高速直连，可接 SDRAM/Flash，最大总线速度为 133MHz。
- 主机接口 HPI：32bit 的 HPI 提供和多种工业标准的主处理器或 PCI 桥芯片相连。
- PCI：32bit 的 HPI 被复合成 PCI 端口，总线频率为 66MHz。DSP 提供配置寄存器控制 PCI 使用，有专门的 EDMA 通道。
- 多路缓存串口 McBSP：McBSPs 能够和多种标准的端口相连，McBSPs 是一种同步串口。Vport0 和 Vport1 与 McBSPs 共用端口。
- 通用 I/O 端口 GPIO：GPIO 提供通用 I/O 端口支持，当配置成输出时，用户能控制驱动状态为输出，当配置成输入时，用户能检测到状态为输入。

DM642 针对视频应用为开发工程师做了大量的准备工作，使其更多关注视频软件、算法的开发研制。同时，DM642 本身是一个 PCI 设备，这为大量的 PCI 多媒体压缩板卡提供了方便，降低了系统成本。可以说 DM642 是 TI 公司全力为多路视频高端应用而打造的方案，并且在业界也已经创造了不菲的业绩。

TI 最新推出的 CPU 为 C64x+的达芬奇技术正在被各大公司研究和研发。它完全兼容 C64x 的代码，DM642 的算法不加改动就可以在达芬奇平台上运行。达芬奇最开始是双核：DSP+ARM，后来达芬奇推出了 C64x+的 DSP 单核平台。达芬奇更是针对数字视频应用而量身定

做的，公司投入开发周期缩短、开发出的产品功能也更强大。

13.2 DSP软件开发环境CCS

由于 DSP 开发难度相对单片机 MCU 要大得多，所以 TI 公司为用户提供了界面友好、功能强大的开发软件 CCS（Code Composer Studio）。它集编辑、编译和连接于一体，支持在线程序调试、算法优化工具组合、实时操作系统 DSP/BIOS 开发等。支持功能插件、基于片上支持库 CSL（Chip Support Library）、eXpress DSP 算法等的系统软件、算法软件开发等。

DSP 软件开发环境 CCS 有两种工作模式：软件仿真（Simulator）模式和硬件仿真（Emulator）模式。软件仿真是 CCS 根据选定的仿真平台利用仿真软件在计算机内部申请一部分空间，模拟 CPU，不和任何硬件打交道，软件仿真特点是没有目标板照样可以编程，缺点是仿真速度慢，无法仿真某些外设的功能。硬件仿真的的是用户把目标板、硬件仿真器和 CCS 连接，仿真程序是下载到芯片中运行的，硬件仿真速度快，仿真结果与实际系统一致，缺点是必须有仿真器和目标板。这两种仿真模式下的 CCS 环境界面基本是一样的。

TI 的 DSP 有三大系列：C2000、C5000、C6000，CCS 均支持 3 个系列的 DSP。目前 CCS 主流版本有 CCS 2.1/2.2/3.1/3.3。CCS 2.2 目前开发人员使用得最多，也较稳定，这里主要介绍 CCS 2.2 版本下的 C6000 系列 DSP。

13.2.1 安装CCS

使用的 CCS 安装程序是 EVM 开发套件自带的光盘 ccs_2.2，安装完 ccs_2.2 后，还需要升级 CCS 以支持 DM642 DSP 的开发，即再安装升级包 CCS 2.20.18。安装的流程比较简单，双击 ccs_2.2\setup.exe，启动安装向导，如图 13-2 所示，接着按照自带的提示完成安装即可。

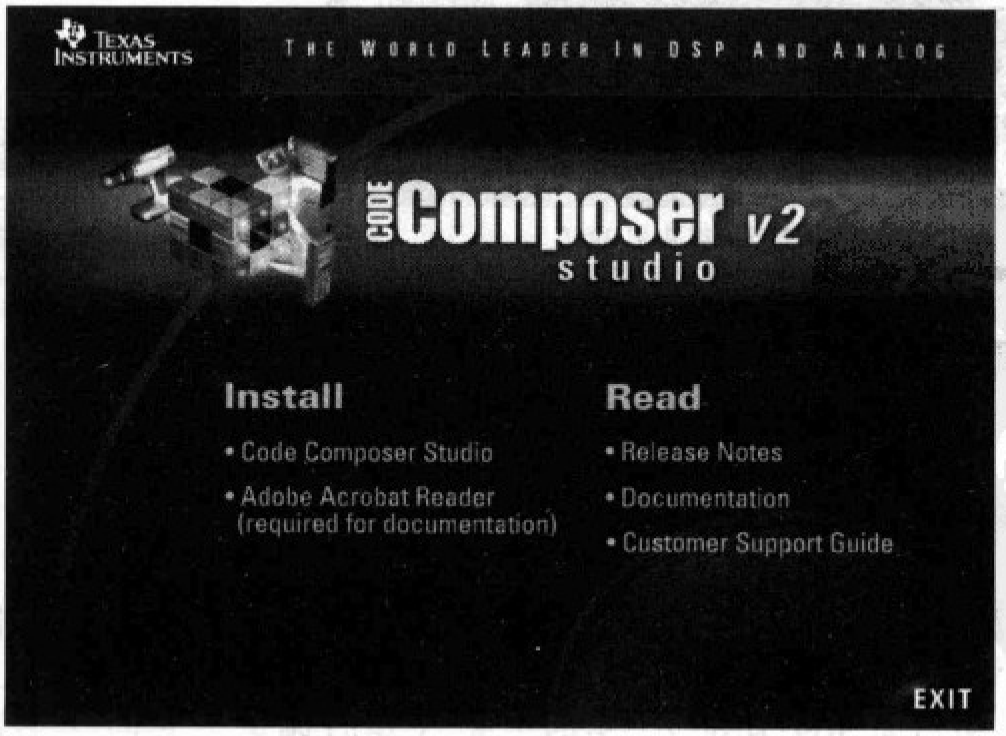


图 13-2　CCS 安装向导

完成 CCS 安装需要重启计算机，运行 D:\ti\dosrun.bat 批处理文件，设置系统环境变量。这样安装软件在桌面上添加了两个快捷方式：Setup CCS 2('C6000)和 CCS 2('C6000)。前者是配置 CCS，包括芯片类型的选择、软仿真、硬仿真的配置程序，后者是 CCS 软件。

至此 CCS 2.2 的安装程序完成，但是原始版本 CCS 2.2 并不支持 DM642 的开发，EVM 评估板自带的光盘中提供了 CCS 的升级包 C6000-2.20.00-FULL-to-C6000-2.20.18-FULL.EXE。现在开始安装，运行光盘目录\EVMDM642\DM642EVM100b\Release\AddOns\CCS2.20.18\下的安装包，按照安装向导完成升级包的安装即可。

下面开始介绍 CCS 的组成部分及有关的开发应用。

13.2.2 CCS主要部件

CCS 2.2 可以支持 TI 的所有系列 DSP，关键看用户安装的 CCS 支持系列，C2000、C5000 和 C6000 可以同时安装。DM642 是 C6000 系列中的 C64x。CCS 的主要部件包括 C6000 代码生成工具、CCS 集成开发环境和 DSP/BIOS 插件，如图 13-3 所示。

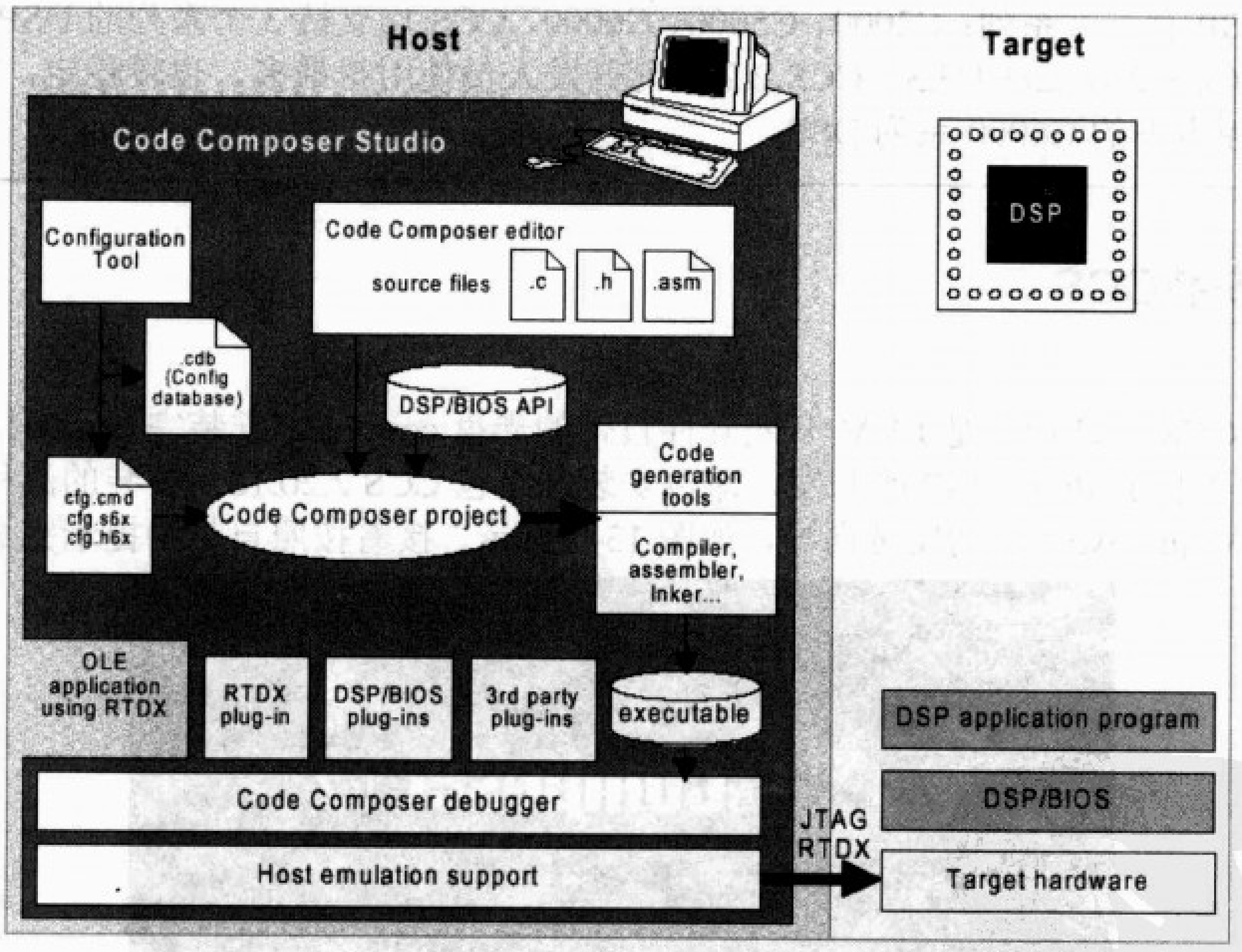


图 13-3 CCS 主要部件

1. C6000 代码生成工具

C6000 代码生成工具具备基本的调试功能，同时还具备实时分析的能力；支持整个软件开发过程：设计、代码生成（项目建立、源程序编写和 CMD 配置文件）、调试（检查语法、设置探测点和日志 log 功能）、实时分析（实时调试、实时状态显示和任务切换跟踪等）。

1）C 编译器

CCS 的 C 编译器支持标准 C 编程。编译器把 C 代码生成 C6000 的汇编代码。C 编译器又可分为语法分析器、C 优化器和代码生成器。TI 的 C 编译器功能非常强大，很重要的一个原因就是它的 C 优化器，它根据编译器的优化选项分别有 4 个不同级别的优化，分别是-o0、-o1、-o2、-o3，优化程度依次递增。C 优化器完成的最重要的优化工作是软件流水（Software Pipeline），从-o2 开始优化器对程序内的循环做软件流水处理。软件流水是针对循环代码如 for 的一种优化技术，软件流水可以生成非常短小的循环代码。

C 语言一级的代码优化还有其他技术，如关键字的使用等，这将在后面的内容中讲解。

2）汇编优化器

C 优化器是针对 C 代码的优化，而汇编优化器的功能则是对用户自编的线性汇编代码（*.sa）进行优化。汇编优化器是 DSP 业界 TI 独具特色之处，它能对用户的线性汇编自动优化成并行汇编程序，这对绝大部分 DSP 开发者降低了开发并行汇编的难度。近乎按照 C 语言编程的线性汇编就能完全达到并行汇编的效果，汇编优化器将用户的线性汇编代码生成并行汇编。线性汇编是为简化 C6000 汇编语言编程而开发设计的，它不是一个独立的编程语言，采用线性汇编一般不需要考虑并行指令安排、指令延时、寄存器使用等，可以任意定义多个寄存器变量，非常类似于 C 语言的编程。

3）汇编器

汇编器把输入的汇编文件生成目标文件*.obj。汇编文件可以是 C 语言编译生成、线性汇编编译生成或库文件内的宏，汇编器产生 TI 的 COFF 格式文件。汇编代码除了 C6400 的机器指令外，还有汇编伪指令，汇编伪指令用于宏定义和宏扩展，控制代码段和数据段的内容和存放地址、预留数据空间、初始化数据、控制编译器优化过程和符号调试等。

4）链接器

链接器的作用是把*.obj 文件生成可执行的目标文件*.out。链接器根据用户说明的程序和数据存放地址把汇编器产生的浮动地址程序和数据映射到用户系统的实际地址空间，用户地址的说明在*.cmd 文件中。把数据和程序的实际地址分配放在链接阶段进行，方便修改，同时有利于程序代码在不同系统间的移植。

2. CCS 集成开发环境

CCS 是一个开放式的插件开发环境，在基本的功能支持下，可以安装其他功能插件，如集成安装了代码生成工具、软件模拟器、实时操作系统 DSP/BIOS、实时分析和数据可视化等功能。前面介绍了 CCS 的安装过程，现在介绍如何使用 CCS 进行软件开发，本书以支持 C6000 系列即 C62x/C64x/C67x 的 CCS 2.2 版本为例。

1）配置 CCS

STEP 01 双击桌面上的“Setup CCS 2 ('C6000)”快捷方式，启动如图 13-4 所示的对话框。用户可以对所有的配置利用特定的过滤器“筛选”出所需要的配置选项，如“C64XX/Simulator/littleEndian”。

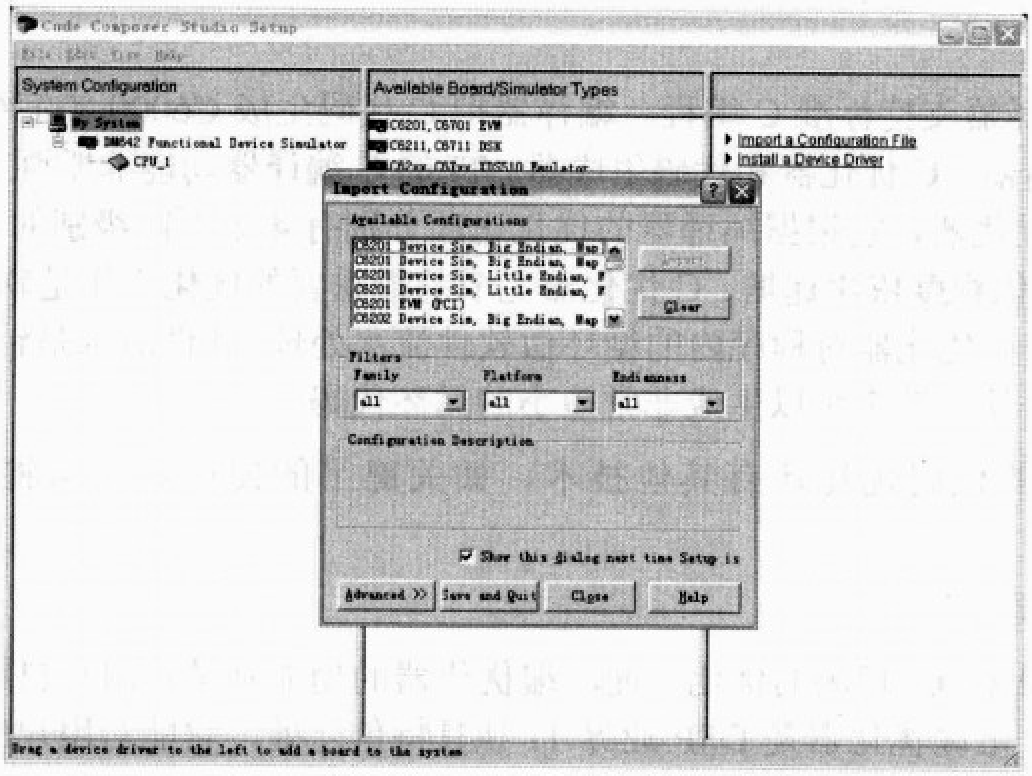


图 13-4　CCS 配置软件

STEP 02 选择“Import Configuration”对话框中的“DM642 Functional Simulator，Little Endian”选项，如图 13-5 所示。下方的“Configuration Description”列出该配置的支持功能，它可以仿真 EDMA 和 QDMA 等功能。单击“Import”按钮导入选中的配置选项，配置完成后如图 13-6 所示。

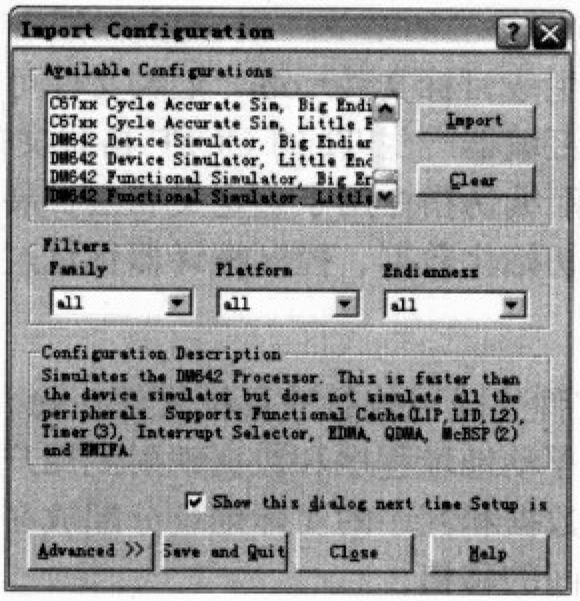


图 13-5　CCS 配置对话框

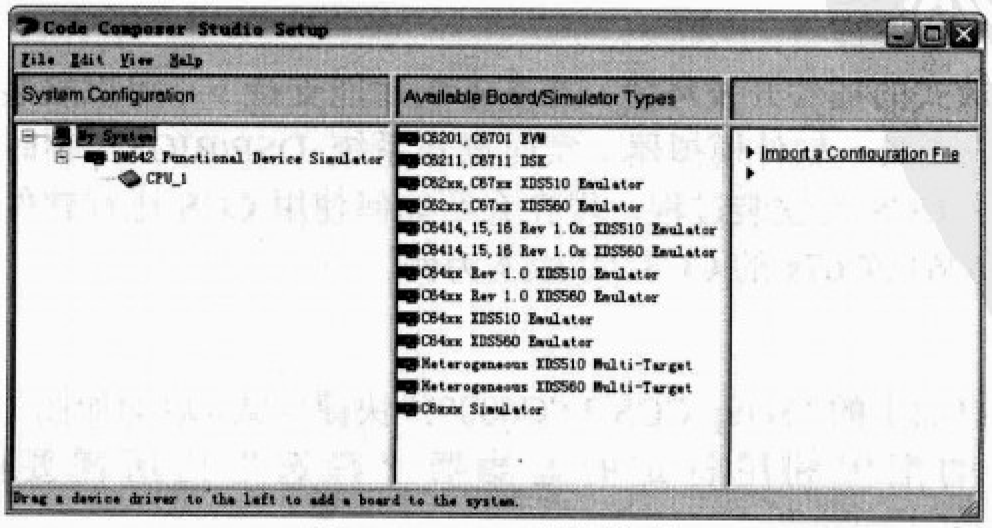


图 13-6　CCS 配置软件

STEP 03 关闭如图 13-6 所示的对话框，弹出如图 13-7 所示的提示对话框，用户确认是否保存更改。单击“是”按钮，激活如图 13-8 所示的画面，用户确认是否现在启动 CCS 软件。

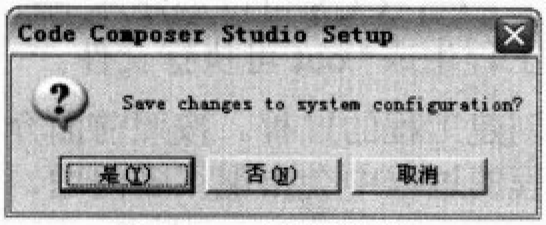


图 13-7　保存设置

图 13-8　启动 CCS

STEP 04 单击图 13-8 中的“是”按钮，立即启动 CCS，如图 13-9 所示。CCS 开发软件与 VC++的界面非常相似，CCS 集成开发环境的使用说明可参考其他有关书籍或 CCS 的在线帮助。文件 D:\ti\docs\pdf\manuals_ccs_full_c6000.html 列出了 CCS 的英文开发手册。

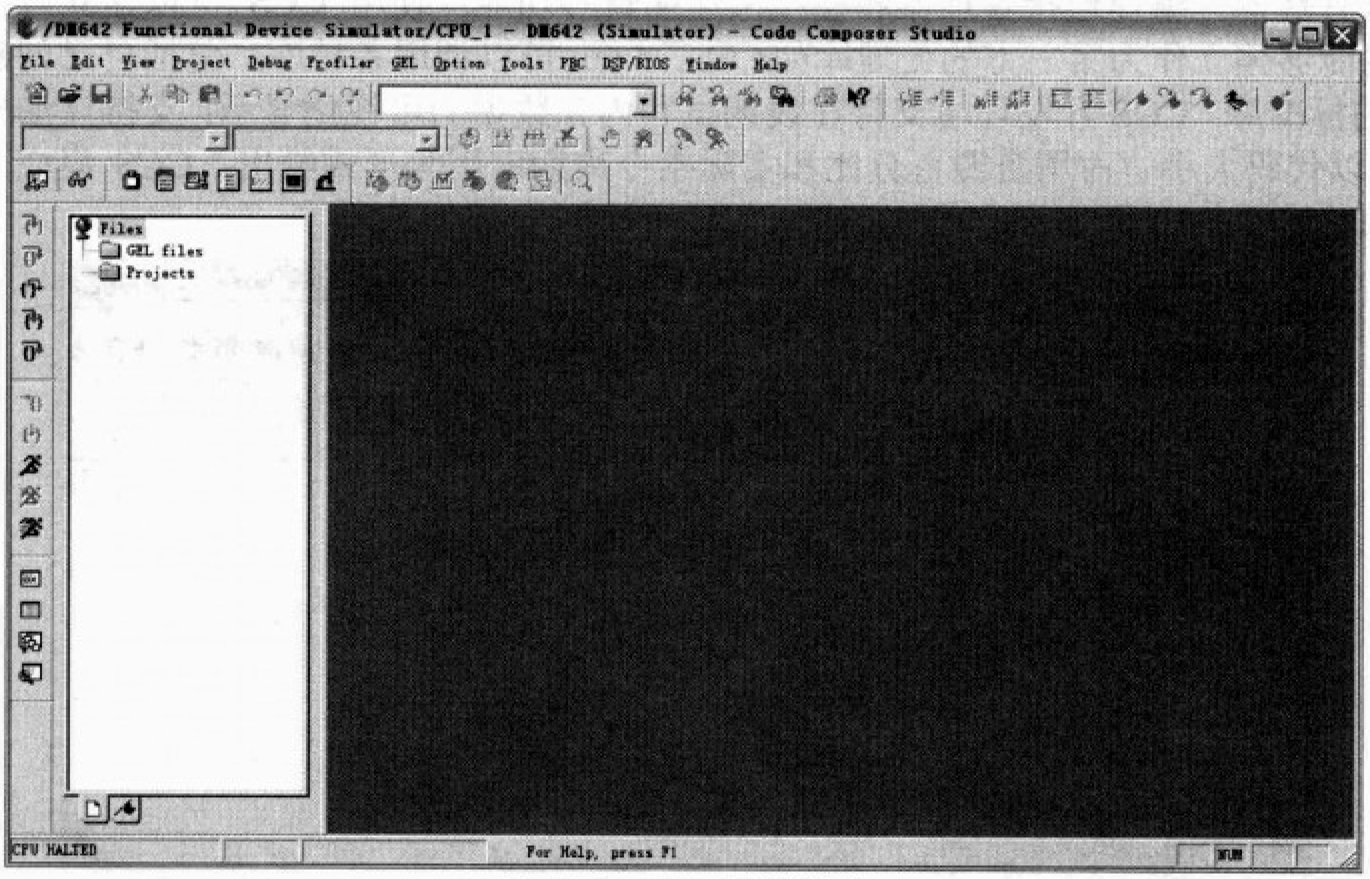


图 13-9　CCS 开发软件

STEP 05 如果开发者使用硬件仿真，也已经在配置 CCS 中安装了相应的驱动，但假若没有连接仿真器或仿真器没有连接目标板，则启动 CCS 都会提示报错，软件仿真则不会存在上述问题。并且，如果配置了两个或两个以上的驱动，如配置了一个硬仿真的驱动和一个软仿真的驱动，则 CCS 启动成功后会出现一个“Parallel Debug Manager”程序，如图 13-10 所示。选择不同的模式或平台仿真，单击“Open”下的相应驱动即可。

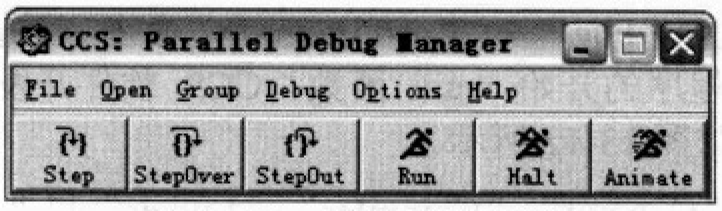


图 13-10　并行调试管理器

上述配置两个驱动一般用于同时进行开发 DSP 系统软件的硬件仿真（Emulator）和开发算法的软件仿真（Simulator），这样就可以同时开启两个 CCS。CCS 的标题栏标出了当前 CCS

的驱动名称，图 13-9 使用的是 DM642 的软仿真。

2）使用 CCS 创建工程

前面介绍了 CCS 的驱动配置，现在使用 CCS 创建工程，CCS 可以创建库工程和应用程序工程。库工程生成链接库*.lib、*.l64 等，应用程序工程生成*.out 可执行文件。

现在使用 CCS 的软仿真（Simulator）介绍 CCS 创建工程的过程。按照前面介绍的过程启动 CCS 2.2 开发软件，在创建库工程和应用程序工程的过程中会出现很多问题，由于篇幅原因，这里不再逐一详细介绍创建过程。而在 13.5 节以库、应用程序工程创建为内容，介绍 DM642 的常见开发问题，读者也可以参考其他 C6000 开发书籍或 CCS 在线帮助。

3）CCS 集成开发环境

使用过 VC 的开发人员对 CCS 的开发环境界面应该非常熟悉。两者很相像：有项目管理器、调试管理器、编译连接管理器和程序快速编辑等功能；使用 CCS 可以建立可执行文件*.out，也能够建立作为第三方的处理库如*.lib。*.out 可以调用各种库，包括自己创建的库和 TI 提供的标准库。CCS 开发环境支持在线调试和内存数据的可视化显示；支持算法资源剖析 Profile，以代码大小、占用资源百分比和实际指令周期占用等形式给出。CCS 集成开发环境如图 13-11 所示。

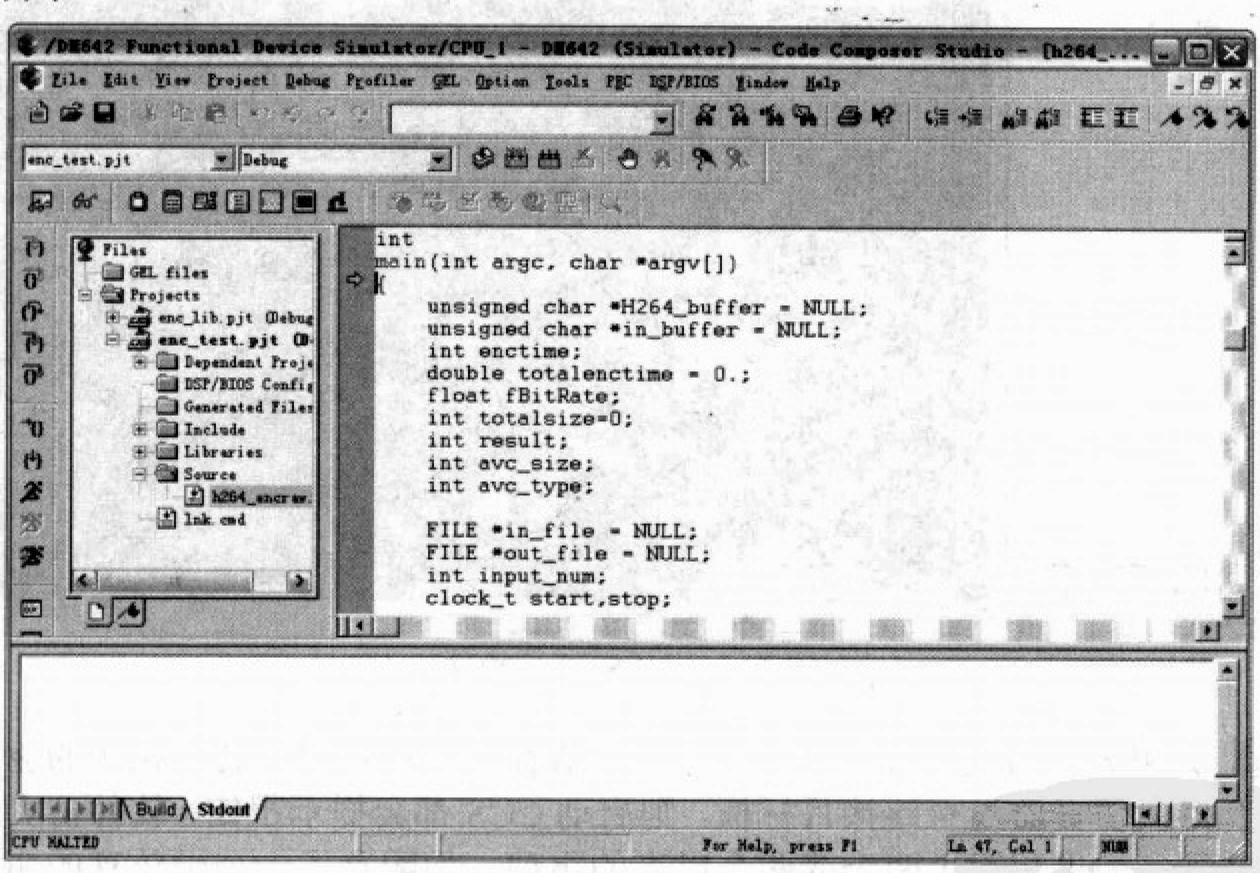


图 13-11 CCS 集成开发环境

编译成功的可执行文件还必须下载到芯片中才能运行，即使是软件仿真也要有这个动作“Load Program”，途径是选择“File”→“Load Program”命令。下载完毕后，单击“Debug/Go Main”按钮，调转到用户程序的开始 main 函数。CCS 软件开发环境最重要的选项配置是“Projects/Build Options”，如图 13-12 所示。

工程编译选项包括了“General”、“Compiler”、“Linker”和“Link Order”选项卡，对工程的所有编译、链接、CCS 优化工具等都在此进行设置。对于库工程，编译选项没有“Linker”选项卡，因为库本身不是执行程序不必链接目标文件或其他库文件。在“13.2.3 使用 CCS 优化工具实现算法优化”一节详细介绍工程编译选项的设置及其功能。用户也可以单击图 13-12 中的“帮助”按钮，激活在线帮助。

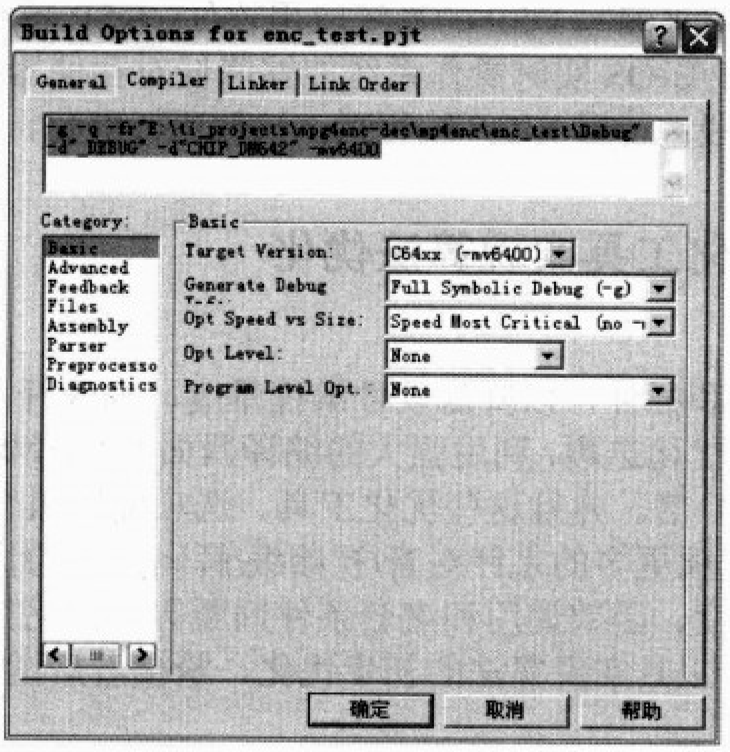


图 13-12　CCS 工程编译选项

3. DSP/BIOS 实时操作系统

音/视频处理硬件应用往往有多个任务：捕获、处理和输出。开发者自己来实时维护这样一个系统难度非常大，资源的共享、音/视频的同步、多通道的管理等都急需一个多任务管理软件，BIOS 就是这样一个管理多任务的实时操作系统。开发人员只负责任务的具体行为，即任务函数体的实现，然后捆绑到 BIOS 的配置中，任务函数间的调度由 BIOS 帮助开发者完成。DSP/BIOS 展现给编程人员的主要是一个可视化的配置界面，一个典型的 DSP/BIOS 功能配置如图 13-13 所示。

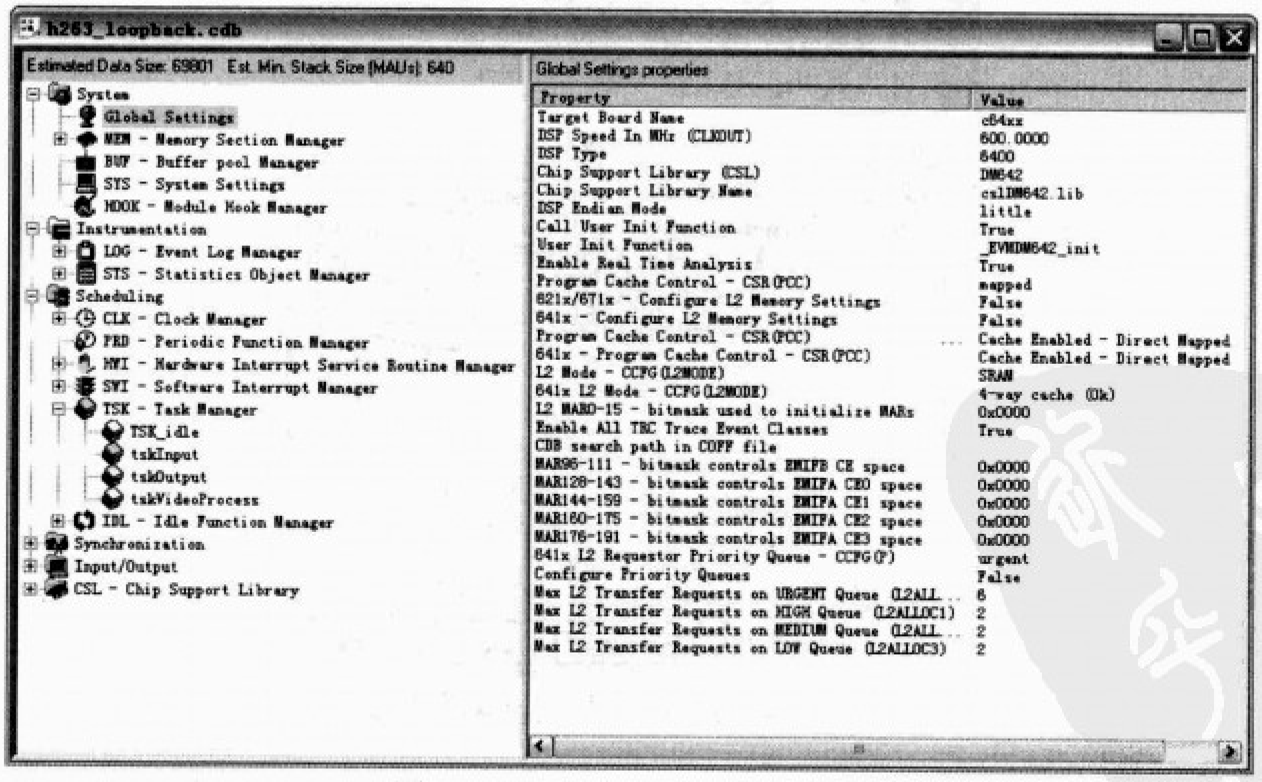


图 13-13　DSP/BIOS 配置实例

DSP/BIOS 主要是为满足实时调度和同步要求，以及主机/目标系统间的通信和实时监测等的应用而设计的。其主要组件包括 DSP/BIOS 配置工具、DSP/BIOS 实时分析工具和 DSP/BIOS 内核 API 函数。

使用DSP/BIOS主要用来编写DSP的系统应用软件：DSP系统的各种资源的维护、调度各种算法或第三方库。DSP/BIOS实时操作系统能够降低对存储器和CPU任务的需求；BIOS提供了多种多任务开发手段；使用BIOS编程后，软件更标准化，可以快速集成第三方软件。

13.2.3 使用CCS优化工具实现算法优化

CCS集成了多种工具和插件，以降低软件编程难度、加快开发进度。特别是C编译器，CCS根据用户提供的编译优化选项，利用强大的编译器如C编译器和汇编编译器能提高不做任何优化算法效率的3～4倍。并且这些优化工具、选项能够组合以满足用户的调试需要。CCS的优化工具和编译选项更多的是针对音/视频编/解码算法的优化，而对于系统软件或应用程序由于其多种任务调度、函数调用和潜套条件判断等，优化工具优化效果不明显。这里主要展示使用CCS的优化工具实现算法的初步优化，紧接着后续两节讲解优化的根本手段：软件编程优化。

CCS的优化工具在CCS集成开发环境的工程编译选项中设置，并且工程的几乎所有设置也在该配置中设置，如图13-12所示的工程编译选项。主要包括产生调试信息选项、程序速度和代码大小选项、优化等级、程序级优化，这些优化编译选项彼此独立，根据需要可以组合出多种优化设置。

1．产生调试信息选项 Generate Debug Info

产生调试信息编译选项如图13-14所示，该选项决定编译出的程序是否支持全字符调试、功能剖析和无任何调试信息等功能。

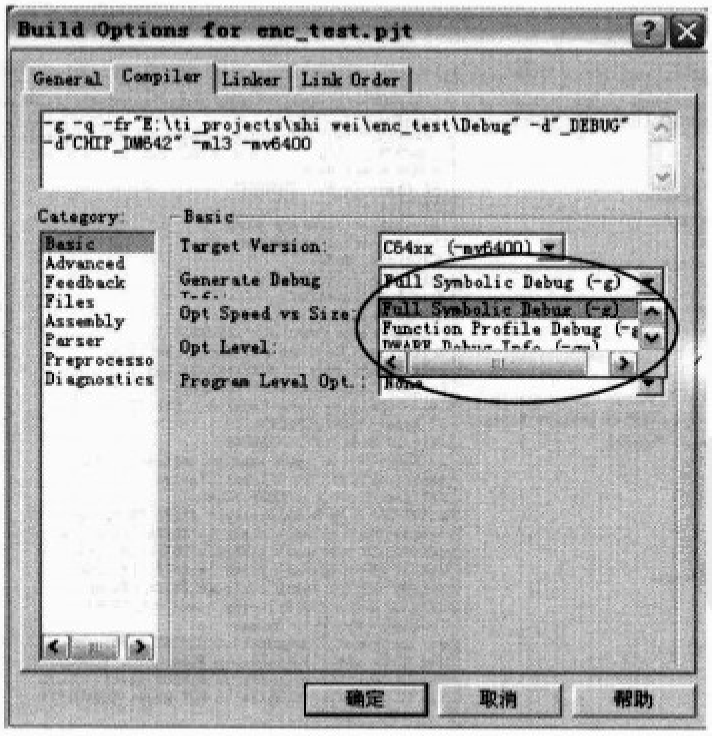


图13-14　产生调试信息编译选项

- Full Symbolic Debug（-g）全字符调试：编译出的程序支持指定语句调试，程序不做任何级别的优化。这种选项一般在最初编写程序的时候使用，程序首先保证结果的正确性。有的时候使用CCS的优化编译选项编译出的程序运行结果错误或程序跑飞，这就需要该编译选项确定程序最初结果是否正确，该选项编译出的程序代码往往很大。

- Function Profile Debug（-gp）功能剖析调试：编译出的程序支持资源耗用和代码大小等功能的剖析，这在开发数据处理算法时，剖析资源占用情况定位核心模块，确定最耗时模块和代码。
- DWARF Debug Info（-gw）产生用于 C/C++源文件、汇编源文件的 DWARF 调试信息。
- No Debug 无调试信息编译选项：编译出的代码不支持任何调试。如果在某语句行设置断点，则运行程序时 CCS 会提示断点无效。因为该选项在编译时编译器做了部分语句调整和无用语句的删减，在最终发布程序时，需要设置该选项。因为一方面编译出的程序代码精简，另一方面编译器做了部分优化，程序效率会有提高。

2. 速度和大小优化 Opt Speed vs Size

速度和大小编译优化如图 13-15 所示。该优化编译选项对程序编译的执行速度和代码大小进行综合优化。编译器为深入优化程序以提高速度，代码会比通常更大。但是有时考虑程序代码不能超过某个限制，否则无法下载和烧写程序，则需要牺牲代码执行速度而使代码尺寸变小。不过，现在的存储器一般都较大，对代码尺寸大小几乎没有限制，而执行速度是最为关注的。

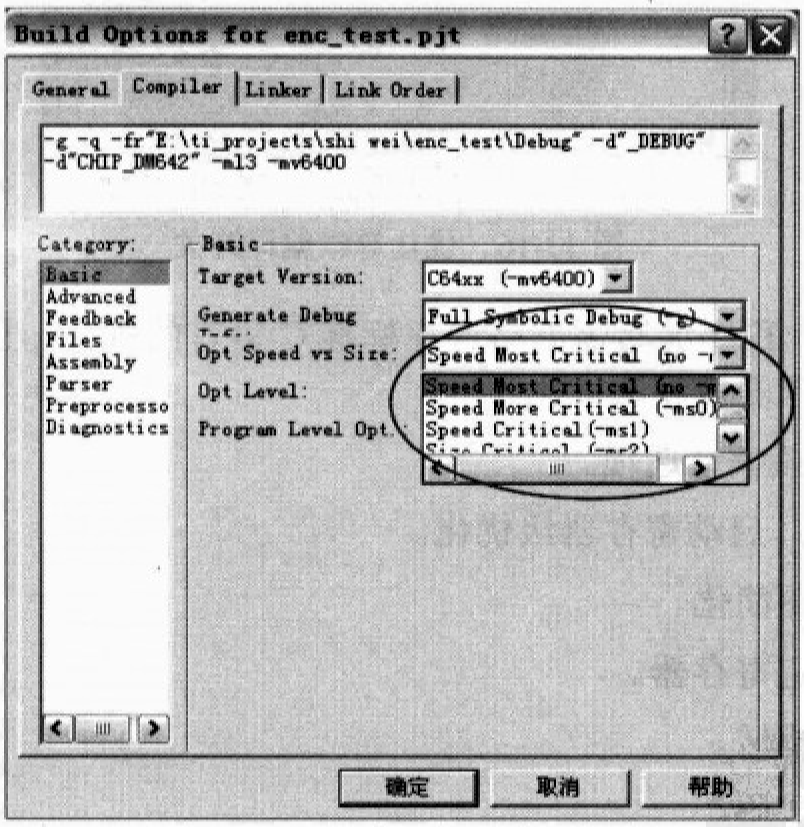


图 13-15　速度和大小编译优化

该编译选项就是对程序执行速度和代码尺寸大小进行综合优化的。

- Speed Most Critical（no –ms）：代码执行速度最重要，此情形代码尺寸往往最大。
- Speed More Critical（-ms0）：代码执行速度比较重要。
- Speed Critical（-ms1）：代码执行速度重要。
- Size Critical（-ms2）：代码尺寸重要。
- Size Most Critical（-ms3）：代码尺寸最重要，即编译出的程序代码最小，此情形代码性能很低。在选择该选项时，多数选择“Speed Most Critical”编译程序代码。

3．优化等级选项 Opt Level（-ox）

优化等级编译开关如图 13-16 所示。CCS 为 DSP 开发者提供算法优化工具主要就是体现在该选项上，选项对算法系统从基本的寄存器、函数到文件级别进行不同程度的优化，CCS 的该选项编译出的程序效率比未使用该编译选项的程序最多可高出 4 倍多。但有时程序太复杂或者编写不规范，“-o3”编译出的程序在执行时结果出错或系统跑飞，因为该编译选项优化程度最强，对程序整体优化彻底。

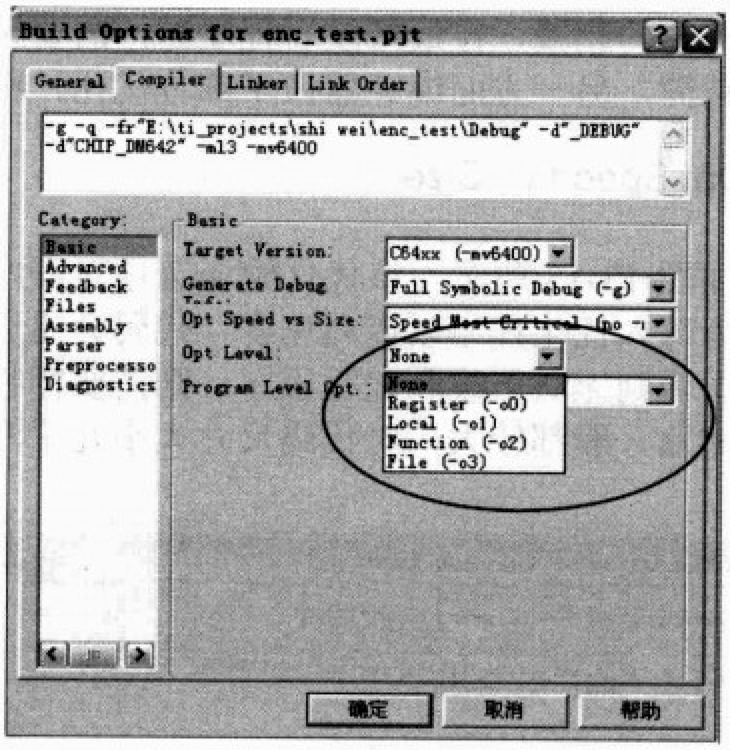


图 13-16　优化等级编译开关

为充分利用该选项，程序编写需严格遵守标准 C 语言的规范，且把算法优化的信息尽可能详细地设置以帮助编译器做算法优化。

- None：不使用该编译选项。
- Register（-o0）：启动寄存器级优化。
  - 执行流控图简化。
  - 给变量指定寄存器。
  - 执行循环旋转。
  - 去除无用代码。
  - 简化定义和语句。
  - 内联展开 Inline 函数。
- Local（-o1）：启动本地优化。默认-o0 的优化等级，同时还支持下列功能。
  - 执行本地变量、常量复制传递。
  - 删除无用的赋值语句。
  - 去除本地一般表达式。
- Function（-o2）：启动函数级优化。默认-o1 的优化等级，同时还支持下列功能。
  - 行软件流水。

> 执行循环优化。
> 去除全局的一般从属表达语句。
> 去除全局未使用表达式。
> 把循环中的数组引用转换为指针形式。
> 执行循环展开。

如果只是用-o 而没有表明优化等级则编译器默认使用-o2。

- File（-o3）：启动文件级优化。默认-o2 的优化等级，同时还具有以下功能。
  > 移除所有从来不使用的函数。
  > 用 return 语句简化未使用的函数。
  > 小函数调用使用内联 Inline。
  > 记录被调用函数名，以便调用函数在优化时能掌握被调函数的属性。
  > 当所有调用函数在同一参数位置传递相同的参数值时，传递参数到函数体内部。
  > 标识文件级变量特征。

为充分利用编译优化工具，提高算法性能，尽量使用-o3 优化选项。

4．程序级优化 Program Level Opt（-pm x）

程序级优化如图 13-17 所示。在程序级将代码优化，该选项允许编译器对整个项目的所有源程序联合观测，程序性能得到优化的同时代码尺寸也将得到优化，只要可能就启用该功能。

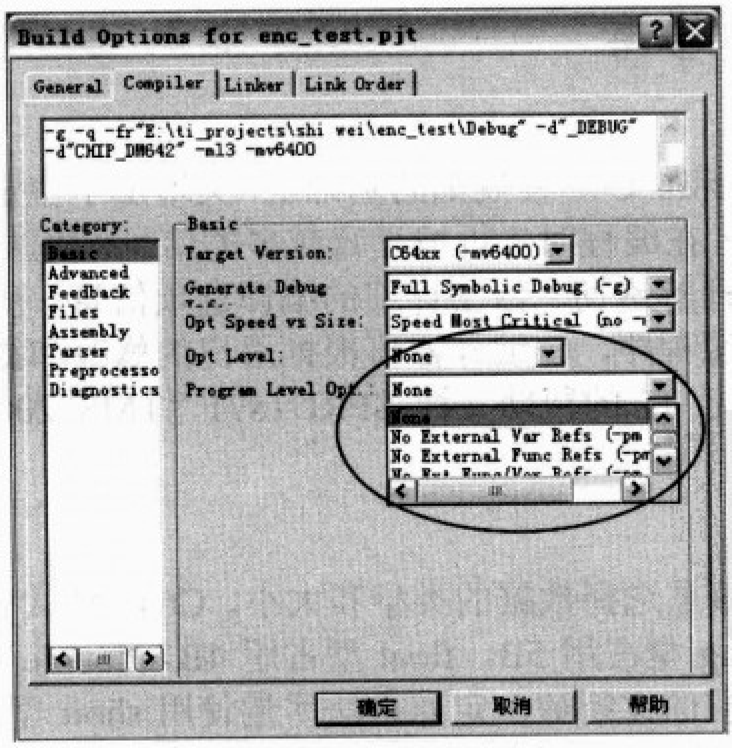


图 13-17 程序级优化

- 无外部变量参考：当前模块没有使用在其他模块定义的变量，命令行参数-op3。
- 无外部函数参考：当前模块没有使用其他模块定义的函数，命令行参数-op1。
- 无外部变量或函数参考：当前模块没有使用其他模块定义的变量或函数，命令行参数-op2。

● 有外部变量或函数：当前模块使用了其他模块定义的变量或函数，命令行参数-op0。

以上是使用 CCS 的编译工具对程序进行性能优化和代码尺寸优化，但是最根本的方法还是软件编程实现算法程序的彻底优化，编译器优化只是辅助手段。

13.3 用C语言进行视频算法框架编程

CCS 的标准 C 语言编程给所有的软件研发工程师带来了极大的方便，同时 CCS 的 C 编译器对编排良好的 C 程序能做到 90%的汇编效率。另外 CCS 在支持标准 C 语言的同时，又增加了额外的 C 语言开发环境下的关键字和优化方法。特别需要提及的是 CCS 提供了 C 语言环境下的汇编指令编程，即 intrinsic 指令，编译器在编译时每条 intrinsic 指令就对应一条汇编指令，完全支持 C 环境下的在线调试，这样可以完全避免汇编编写的烦琐规则和调试不友好的问题。intrinsic 有些类似于一个 C 语言的功能函数，但是它不占用额外的压栈、弹栈等函数调用操作指令周期。并且绝大部分汇编指令都有对应的 intrinsic 指令，编译程序的效率和纯汇编相当。几乎所有的系统和视频算法在编写时都采用 C 语言编程，并且开源的视频 CODEC 也几乎都是 C 语言实现。视频算法框架主要完成编/解码器的创建、销毁，处理数据的读取、处理结果的输出等模块。根据 CCS 下的 C 语言编程规则给编译器设置优化信息、编排视频算法框架和修改数据结构。

大家知道 VC 2005 既支持 C++编程又支持标准 C 编程，CCS 类似于 VC 2005，两种语言均可采用。但考虑算法编译效率和算法平台移植的因素，建议采用标准 C 编程。

13.3.1 C编程规则和参考

CCS 开发环境在支持标准 C 编程规则的基础上，又增加了特定规则和编译信息以帮助编译器高效编译算法。开发者在编程时告诉编译器越多的信息，算法优化程度越深、效果越好。开发者应尽可能地把算法理解透彻、给出详细的编译选项信息。所以需要用户充分利用编译器反馈的信息不断地修改源程序，C 编译器再根据用户的修改和提供的信息进行优化，这是一个多次交互的过程。详见 TI 的帮助文档 SPRU1871“TMS320C6000 Optimization C/C++ Complier User’s Guide”。

1）数据类型

在定义变量前，需要慎重选择数据的类型和大小。C64x 默认 char 型占用 1B，short 型占用 2B，int 型占用 4B，long 型占用 5B，float 型占用 4B，double 型占用 8B。选用类型时需要注意 int 和 long 占用不同的比特数；定点乘法尽量使用 short 型数据；循环计数器使用 int 型不使用短整型 short。数据变量能使用 short 型就不使用 int 型，为数据打包处理做准备。

2）存储器的相关性

为使编译出的程序具有高效的性能，编译器将尽可能把指令编排以并行执行。为指令能并行，编译器首先要确定指令间的先后关系，或者说确定指令间的相关性。只有那些不相关的指令才能并行起来，相关的不能并行，否则结果不正确。编译器判定的相关性原则是，不

能确定相关性的默认相关，指令串行执行；若能确定指令是不相关的，则安排并行执行。不告诉编译器相关信息，让其单独确定计算结果、存储器访问是否相关则很困难。因此需要开发者人为干预，给编译器提供一些帮助信息。方法有使用 restrict 关键词定义内存指针、联合使用-pm 和-o3 编译选项、使用-mt 选项。

3）数据打包处理

数据打包是 C64x DSP 对多媒体处理的强有力支持。大家都知道，图像像素点多数是单字节的，要循环处理像素块，如果在一条指令内同时运行多像素点的运算，则能极大地提高计算效率。如 ADD4 指令就是对两个 4 个字节分别做有符号加法，显然效率提高了 3 倍。还有 SUB2 是两个整型数做 16 位减法。这样做的前提是处理的数据长度不能超过单字节如 char，或双字节如 short。视频图像数据基本是定义单字节，DCT 变换系数、系数量化结果定义双字节。

4）C 编程参考

在 TI 的$(Install_dir)\C6400 目录下，各种图像处理库 imglib 和 DSP 库 dsplib 提供了许多实用的例程，许多都是进行 C 语言编程的良好参考和编程模板。

13.3.2 DSP关键字与CMD文件使用

DSP 的 C 语言编程使用标准 C，同时还有自己的一些特殊的关键字以帮助编译器更好地编排、优化源程序。另外，DSP 的可执行程序是下载到芯片中运行的，函数、代码的存放位置、内存空间的使用、库的链接等需要专门的一个文件，使程序在下载过程中按照用户的要求放置代码和使用内存，这就是 TI 提供的 CMD 文件格式。

用户可以自己创建 CMD 文件，DSP/BIOS 也会自动生成 CMD 文件。当使用 BIOS 时，典型的使用 CMD 的方法是在自己的 CMD 文件中首先把 BIOS 生成的 CMD 文件链接进来，然后再添加自己的特定使用过程。

1．DSP 关键字

使用 DSP 的特殊关键词，提供更详细的编译信息给编译器可以帮助其编译出高性能的程序。这些关键字主要在 C 语言环境下使用，常用且对代码优化有极大作用的关键字如下。

1）#pragma CODE_SECTION

DSP 的程序和数据默认放置的位置是片外的 SDRAM。该关键字表明某函数体代码放置的位置，为提高指令 Cache 的命中率，需要把核心代码模块放置在 DSP 片上的 L2 SRAM 上，则可以使用该方式。例如，#pragma CODE_SECTION（my_func,".my_sect"）表示函数 my_func 在段.my_sect 中。

2）#pragma DATA_SECTION

类似于 CODE_SECTION，该关键字是把重要数据空间映射到某位置。同样为了提高 CPU 的读写速度，需要把经常使用的数据空间放置在片上 L2 的 SRAM，如待编码的图像一般把一行宏块（16 行）映射到 L2 上，以提高 CPU 的访问速度。例如，#pragma DATA_SECTION（my_data,".my_data_sect"）表示段.my_data_sect 在 CMD 文件中映射到 L2 空间。

3）#pragma DATA_ALIGN

由于 DSP 的 Cache 访问有边界和吞吐量限制，另外 CPU 指令有边界对齐和非对齐指令区别。为了提高 Cache 命中率和 CPU 的访问速度，极力推荐 CPU 访问的内存空间是 8、64、128 字节对齐。例如#pragma DATA_ALIGN（my_data, 128），地址空间 my_data 以 128 字节对齐，推荐 128 字节对齐。

4）#pragma MUST_ITERATE

前面曾提及，告诉给编译器的信息越则编译出的程序性能越高。另外算法中耗时模块多数是循环模块。所以在优化 for 循环时由于是定循环，可以使用关键字 MUST_ITERATE 来通知编译器循环的次数。例如#pragma MUST_ITERATE（min, max, multiple），min 表示循环迭代的最少次数，max 表示循环迭代的最多次数，multiple 表示循环迭代的最小公倍数。

5）restrict 关键字

为了确保指令的不相关，从而能够多条指令并行，需要确保内存访问的不相关性，如一块内存不能在读之后马上进行写操作。方法有使用关键字 restrict 定义指针，声明空间的独立性；把两块内存映射到不同的段或 bank。特别是图像块的变换、量化等整体操作，确保数据源和目标地址是独立地分布在两个段或 bank 中的，例如：

```
Void vecsum(short *restrict sum, short *restrict in1, short *restrict in2, int N)
{
 int i;
 for ( i=0; i<N; i++)
        sum[i] = in1[i] + in2[i];
}
```

在上述例程中，分别声明了 3 个空间为独立的内存段，这样编译器在优化时能够尽可能地编排并行指令，性能大大提高。

2. 使用 CMD 文件

程序在运行的时候是按照预定的存放位置运行的，数据也是根据预定的位置去读写的。DSP 的内存是统一编址的，则在装载可执行程序（Load Program）时，需要有个文件说明代码段和数据段的位置“说明书”，CMD 文件就是来做这个工作的。CMD 文件定义段的名字和段的映射位置及进行静态库的链接等，一个典型的 CMD 文件如下。

```
MEMORY
{
 ISRAM: o = 0x00001000   l = 0x00020000   /* 定义在L2空间上的起始地址和长度*/
 SDRAM: o = 0x80000000   l = 0x02000000   /* 定义在SDRAM上的起始地址和长度*/
}
SECTIONS
{      /*  使用C语言编程的标准段映射  */
       .text        > SDRAM       /* 代码段，程序默认放置的位置*/
       .stack       > SDRAM       /* 堆栈段 */
       .bss         > SDRAM       /* 数据段，未初始化 */
       .cinit       > SDRAM       /* 标准C初始化 */
```

```
        .cio         > SDRAM        /* 基本的输入输出段 */
        .const       > SDRAM        /* 常数段*/
        .data        > SDRAM        /* 数据段*/
        .switch      > SDRAM        /* switch查表段*/
        .sysmem      > SDRAM        /* 系统段*/
        .far         > SDRAM        /* far段声明的全局变量和数组*/
        /* 用户自定义的段映射，为提高CPU读写速度，所有段映射到了L2空间 */
        .mc_sect   > ISRAM
        .dct_coeff_sect > ISRAM
        .qnt_coeff_sect > ISRAM
        .cur_sect > ISRAM
        .ref_sect > ISRAM
}
/* 程序链接的库 */
-l cslDM642.lib
-l rts6400.lib          /* C and C++ run-time library support */
-l img64x.lib
```

上述 CMD 文件是一个典型的在不使用 BIOS 编程系统软件时的案例。如果使用了 BIOS 编程，而用户又需要增加自己的存储器映射，则首先用户自己定义一个 CMD 文件，然后在其中把 BIOS 自动生成的 CMD 文件链接进来即可。用户在程序中使用的段如代码段和数据段的映射就需要自己在 CMD 文件中明确存放位置，如.mc_sect 段是图像做运动补偿时用到的临时空间，为了提高 CPU 的执行速度，案例把它映射到了 L2 上的 ISRAM 空间。其他的重要空间如 DCT 变换结果、量化结果等的空间也映射到了 L2 上。如果源程序中定义了段而在 CMD 文件中没有映射，则 CCS 编译器默认映射到 SDRAM 上。

13.3.3 算法系统资源剖析

在保证算法执行结果正确的前提下，需要开始考虑算法效率优化的问题。算法系统往往是对数据做循环的处理，如整帧图像的以块为单位的编码或解码；写码流到内存、文件调用次数也会较多；其他创建、初始化模块仅仅调用一次等，这些任务开发者需要掌握其消耗资源的占用比，以准确确定资源占用分布情况。这就需要剖析算法系统的资源：CPU 指令多少、代码大小等的分布怎样。CCS 开发软件为工程师提供了信息丰富的算法剖析工具。

STEP 01 将要剖析的工程编译选项 Build Options 修改为 Function Profile Debug 模式，编译整个工程。

STEP 02 在源文件中包含头文件 time.h。

STEP 03 读取当前系统时间和算法运行后系统时间。

```
#include <time.h>
#include <stdio.h>
void main()
{
    clock_t start,stop,over_time;                //定义的时钟变量
    start = clock();                             //读取当前时钟
```

```
    my_process_function();                          //算法处理函数
    stop = clock();                                 //读取当前时钟
    over_time = stop-start;                         //计算算法的处理消耗时间
    printf("Function Used = %d \n",over_time) //打印输出
}
```

上述程序是典型的测试算法函数的消耗时间的代码。不过，这种方法包括了函数 clock() 本身的消耗时间，over_time 就是用户模块消耗的 CPU 指令周期。

STEP 04 启动系统时钟，选择“Profiler”→“Enable Clock”命令，如图 13-18 所示。

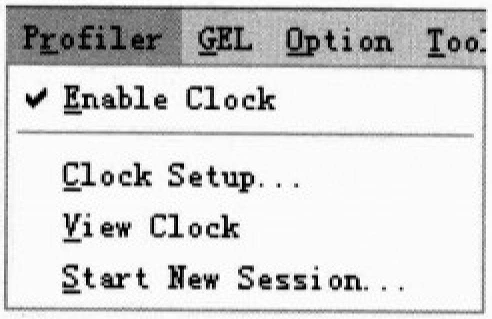


图 13-18　CCS 程序剖析设置

STEP 05 启动新的剖析任务，选择“Profiler”→“Start New Session”命令，如图 13-18 所示。

STEP 06 在两次读取系统时钟的语句处设置断点。方法为首先把光标停在预设置断点的行，然后按快捷键“F9”，这样该行前就添加了一个断点。

STEP 07 编译、下载程序*.out 文件。按“F5”键，执行到第一个断点处。

STEP 08 在图 13-19 中，单击 按钮以激活“Profile All Functions”，对所有函数模块进行剖析。

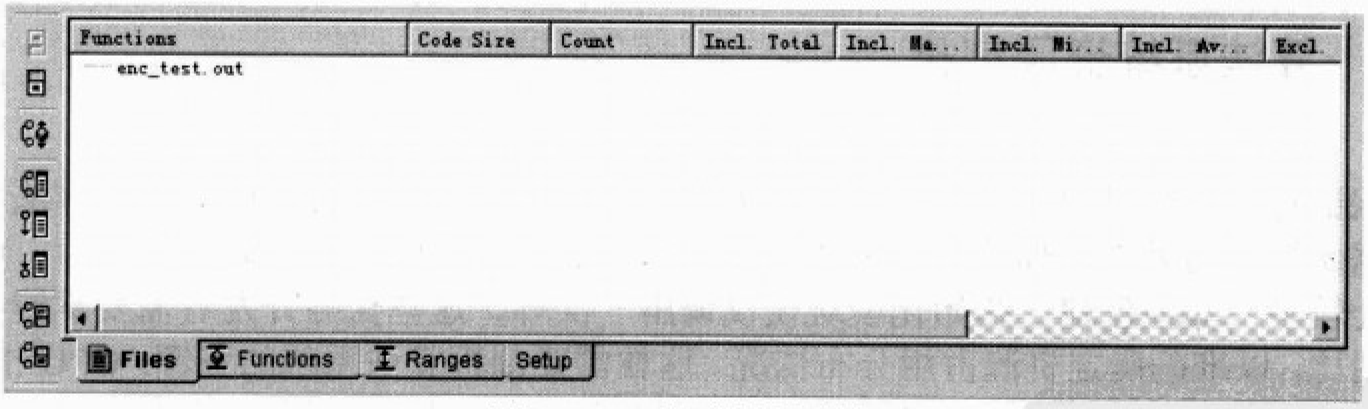
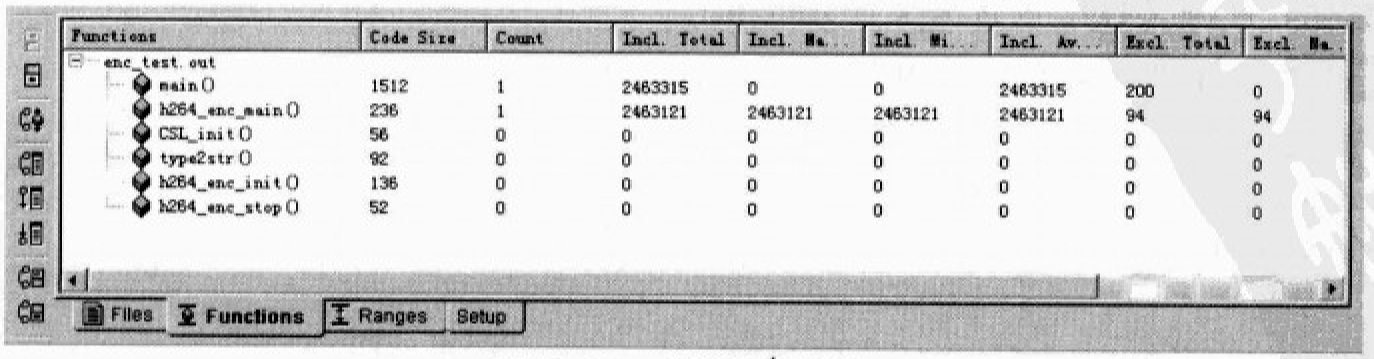


图 13-19　程序剖析清单

STEP 09 按“F5”键开始运行、剖析程序，如图 13-20 所示，该表列出了函数的名称、代码大小、被调用次数、包含其他函数调用的总指令周期及最大和最小指令周期等。



Functions	Code Size	Count	Incl. Total	Incl. Ma...	Incl. Mi...	Incl. Av...	Excl. Total	Excl. Ma...
enc_test.out								
main()	1512	1	2463315	0	0	2463315	200	0
h264_enc_main()	236	1	2463121	2463121	2463121	2463121	94	94
CSL_init()	56	0	0	0	0	0	0	0
type2str()	92	0	0	0	0	0	0	0
h264_enc_init()	136	0	0	0	0	0	0	0
h264_enc_stop()	52	0	0	0	0	0	0	0

图 13-20　剖析资源清单

用户可以单击某个字段如“Inc. Total”，对所有函数的总指令占用排序，随着函数也进行了排序。这样运行到第二个断点处后，CCS 剖析完算法，指令周期的占用比例就一步了然，优化算法时就可以首先从资源占用最高的函数入手，分析、优化算法。

13.4 DM642实现视频算法优化

CCS 开发软件的优化工具的优化结果多数不能满足项目需求，优化过程往往是先采用软件编程的算法优化，最后再采用优化工具的辅助优化。所以视频算法优化的主要手段还是通过编程实现，建立软件流水使多条指令尽可能并行，编写线性汇编优化核心模块，使用 DMA 双缓冲图像数据以提高 Cache 的命中率等，这些都需要用户编程来实现。

算法优化的一般过程是首先使用 DMA 双缓冲图像到 L2 的 SRAM，然后对 L2 上的图像做编/解码，核心耗时模块使用并行汇编或线性汇编优化、改写，最后使用 CCS 优化工具优化算法系统。

13.4.1 并行算法指令和建立软件流水

DSP 的高性能处理的主要原因就是指令的并行执行，TI C64x 系列的 DSP 可以达到 8 条指令的并行，即单指令周期执行 8 条指令。尽管算法系统整体不可能所有指令都是 8 条并行执行，但开发者应尽可能保证更多的指令能够并行，如系统整体平均 6 条或 7 条。视频算法框架由于有 CALL 函数调用、条件判断和 switch 查表等而程序无法进行指令并行。C6000 的所有汇编指令都是条件指令，即指令是否真正执行的判断并不占用额外的指令周期，这对 if-else 指令优化特别有用。

视频算法的核心是对图像块做像素点循环（8×8），CCS 针对这种定循环 for 语句提供了无硬件开销的软件流水（Software Pipeline）。循环条件判断不再占用指令，流水有填充（Prolog）、循环核（Loop Kernel）和排空（Epilog），即建立流水循环、使用循环和释放循环 3 个步骤。在循环嵌套中，只有最内层循环才能进行软件流水。在编排程序进行软件流水时，需要注意以下几点。

- 软件流水循环可包含 intrinsics 指令，不能包含函数调用，但能包含 Inline 函数。
- 循环体不可以有条件终止、使循环提前退出指令如 break。
- 循环必须采用递减计数的形式，编译选项-02 和-03 就能够把尽可能多的循环转换成递减计数的循环。
- 如果在循环体中修改循环计数，则这个循环就不能转换成递减循环。
- 不能在条件语句中增大循环控制变量，否则循环不能进行软件流水。
- 循环体代码尺寸不能太大。C64x 有 64 个寄存器，尽管在使用线性汇编编程时可以自己定义远多于 64 个的变量名称，但循环体不能过于复杂。否则代码不能进行软件流水，可以简化循环或者将循环拆成几个小循环。
- 循环体中要求一个寄存器的生命太长，则代码不能进行软件流水。

- 如果循环体内有复杂的条件代码，则超过 C64x 的 6 个寄存器时，这个循环也不能够进行软件流水。

为了循环体能够实现流水，可以根据编译器编译的汇编结果文件提示，进行有针对性的多次修改。其中一个总体原则：A 和 B 两边的寄存器使用均衡、循环迭代次数尽量少，这样建立的流水效率更高。

CCS 开发环境的编译选项除了能够对整个工程进行设置外，还可以对独立的源文件进行不同的编译选项设置。因为有时因为源文件的不规范或嵌套多层循环，使用-o3 选项编译出的程序在运行时结果出错。这时就需要对该源文件使用低级的优化选项如-o2、-o1。对独立源文件的编译选项设置如图 13-21 和图 13-22 所示。

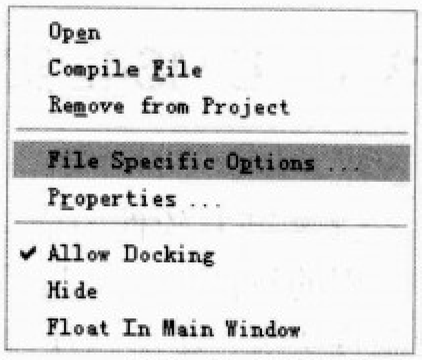


图 13-21　源文件右键快捷菜单

查看独立源文件的汇编编译结果文件，按如图 13-23 所示的进行设置，保留产生的.asm 文件。

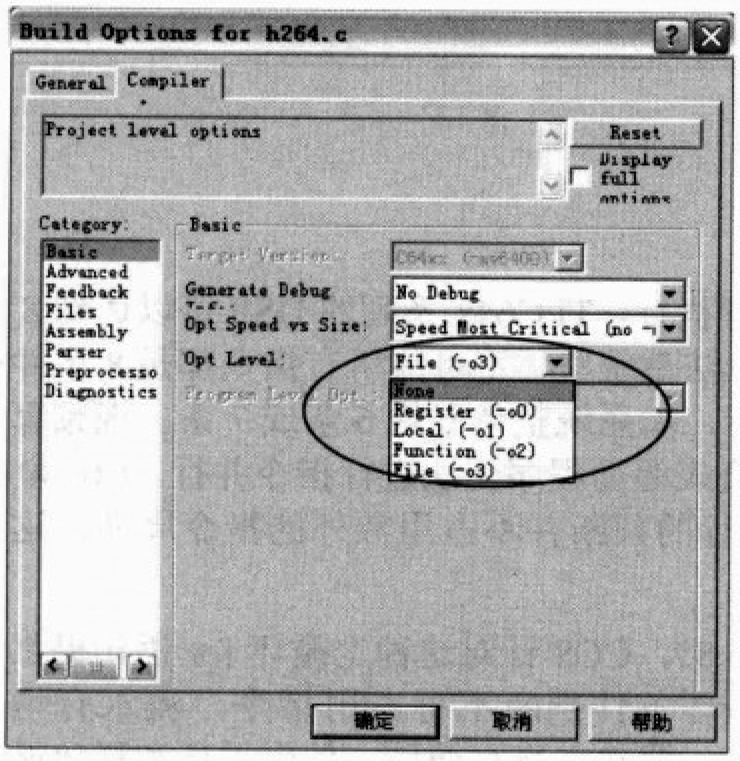


图 13-22　独立源文件编译选项设置

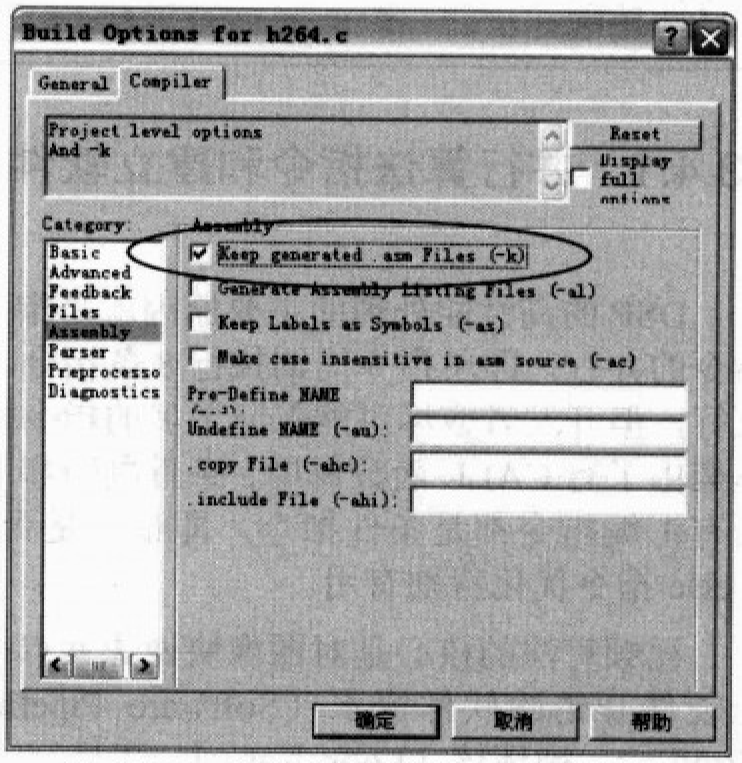


图 13-23　保留汇编编译结果文件设置

下面是对一个线性汇编文件进行并行汇编结果的信息提示，根据提示了解程序的并行程度、寄存器使用情况等。

```
;*----------------------------------------------------------------------*
;*      Loop source line                       : 23  ;
;*      Loop closing brace source line         : 40  ;
;*      Known Minimum Trip Count               : 4   ;已知的最小循环执行次数
;*      Known Maximum Trip Count               : 4   ;已知的最大循环执行次数
;*      Known Max Trip Count Factor            : 4   ;已知的循环次数的偶次分解，即最大迭代
                                                      因子
;*      Loop Carried Dependency Bound(^)       : 1   ;循环传递相关性界限
;*      Unpartitioned Resource Bound           : 3   ;在不划分A/B两侧资源分配前，确定的资
                                                      源边界
;*      Partitioned Resource Bound(*)          : 3   ;在划分A/B两侧资源分配前，确定的资源
                                                      边界
```

```
;*      Resource Partition:
;*                                    A-side   B-side
;*      .L units                      0        0       ;L功能单元，没有使用
;*      .S units                      1        0       ;A侧使用了S功能单元
;*      .D units                      3*       3*      ;A/B均使用了D功能单元
;*      .M units                      0        0       ;
;*      .X cross paths                2        2       ;A/B均使用了X功能单元
;*      .T address paths              3*       3*      ;
;*      Long read paths               0        0       ;
;*      Long write paths              0        0       ;
;*      Logical  ops (.LS)            4        4       (.L or .S unit)
;*      Addition ops (.LSD)           0        0       (.L or .S or .D unit)
;*      Bound(.L .S .LS)              3*       2
;*      Bound(.L .S .D .LS .LSD)      3*       3*
;*
;*      Searching for software pipeline schedule at ...
                                                     ;对循环流水编排反馈的信息，表示以迭代
;*      ii = 3  Schedule found with 4 iterations in parallel
                                                     ;间隔3进行软件流水编排成功。ii越小，
;*      Done                                         ;循环执行的时间越少
;*      Epilog not removed
;*      Collapsed epilog stages       : 0            ;流水填充没有删除
;*
;*      Prolog not entirely removed                  ;流水排空没有全部删除
;*      Collapsed prolog stages       : 2
;*
;*      Minimum required memory pad : 0 bytes
;*
;*      Minimum safe trip count       : 3            ;最小安全迭代次数为3
;*----------------------------------------------------------------------*
```

对定循环 for 应尽可能地编排使之能够软件流水，并检查流水的状态以确保循环核（Loop Kernel）尽可能多的指令并行，且最小迭代次数越少越好，可以说软件流水是算法优化的最重要的技术。

13.4.2 使用intrinsic指令完成核心模块的优化

CCS 开发软件为习惯 C 编程的开发者提供了 C 环境下使用汇编指令的技术，即 intrinsic 指令。C 编译器编译 intrinsic 指令时基本是直接翻译成一条并行汇编指令的，避免了并行汇编中记忆寄存器的抽象名字 A1、B2 等，而是使用 C 环境下定义的变量作为 intrinsic 指令的参数。绝大部分并行汇编指令有对应的 intrinsic 指令，少部分没有。所以开发者在优化核心模块时，可像调用一个函数那样来使用一条 intrinsic 指令。每条 intrinsic 指令都是以下画线开始的，如_add4 表示 4 个字节分别做有符号加法。C64x 的所有 intrinsic 指令详见其指令手册。

```
static void predict_16x16_h( uint8_t *src, int i_stride)
{      //把1字节扩充到一行16字节
 int i;
      for( i = 0; i < 16; i++ ) {
       uint8_t v;
       uint32_t v2,v4;
       v = src[-1];                          //读取1字节
       v2 = _pack2(v,v);                     //把低16位提取形成一个新32位
       v4 = _packl4(v2,v2);                  //把两个V2的偶位字节提取形成一个新32位
       _amem4(&src[0]) = v4; _amem4(&src[4]) = v4;     //内存对齐的写操作
       _amem4(&src[8]) = v4; _amem4(&src[12])=v4;
          src += i_stride;                             //修改源地址，更新到下一行
       }
}
```

上述代码实现的是 H.264 视频编码算法中的亮度块 16×16 的水平方向的预测函数。首先把块左边的一个像素值打包成 4 字节，然后存放回图像的一行 16 个像素，循环 16 次。这样就把块 16×16 的左边一列的像素值扩展到了当前 16×16 块的空间。

13.4.3 使用DSP线性汇编优化核心模块

instrinsic 指令不能指定功能单元，且有的汇编指令没有对应的 intrinsic 指令，编译出的汇编程序有时不能达到预期的优化指标。但是由于 CPU 的寄存器数量多、功能单元多，使用并行汇编编程难度极大。为了达到并行汇编的编译效率，而避免抽象寄存器的使用，TI 为开发者提供了其特有的线性汇编编程语言，它是介于 C 语言编程和并行汇编语言之间的一种汇编语言。用户自己任意定义寄存器名称，且没有 64 的数量限制；编程中可以指定功能单元；能够使用条件指令编程；完全使用并行汇编语言的指令。编排优秀的线性汇编经汇编器汇编出的程序效率和并行汇编语言编写的程序效率相同。TI 也极力推荐 DSP 开发者使用线性汇编语言编程来优化核心模块，再经汇编优化器优化，根据汇编结果提示再修改线性汇编。

线性汇编的语言格式与汇编语言相同，扩展名是*.sa。指令名称采用助记符的形式，另外还有自己的伪指令如.cproc、.endproc 等。线性汇编中功能单元在选择的时候需要慎重，原则是尽量指明，但要保证正确。在开发线性汇编编程时，需注意以下几点。

- 用户可以指定具体的功能单元，如：

```
LDW .D2 *a_src, a_3210 ; 读取地址a_src的内容到a_3210
```

上述指令使用的是 D 单元，寄存器均为 B 侧。

- 在指定功能单元的基础上还可指定是哪一侧的寄存器，下面的指令 D1 产生地址，T2 指定 B 侧寄存器。

```
LDW .D1T2 *a_src, a_3210; 读取地址a_src的内容到a_3210
```

- 用户可以仅指明功能单元类型（.M），而不指明具体哪个单元（.M1），具体由汇编优化器来安排。

● 用户也可以仅指明数据通道或单元，如.2 表示 B 侧数据通道，由汇编优化器根据助记符功能来安排功能单元。

如果开发者分配资源不当，则会限制优化器的优化工作。另外，使用伪指令.reg 来定义寄存器，数目不限制。线性汇编的伪指令有多个，它用来帮助汇编优化器对线性汇编程序段做优化。

下面以两个例程展示使用线性汇编优化的过程，内存图像的读取并做类型扩展的函数 transfer_8to16copy_c，Intra 块的 H263 量化函数 quant_h263_intra_c。前者处理的数据是整个块，而后者处理的数据是除去第一个单元的块。一个是整块数据，一个是非整块数据。这是视频编码算法中两种典型的处理方式。

1. ransfer_8to16copy_c

函数功能是把内存的图像复制到一个 short 类型的数组，复制 8×8 个像素点，为 DCT 做数据源准备。

```
void transfer_8to16copy_c(int16_t * const dst,
                          const uint8_t * const src,
                          uint32_t stride)
{
 uint32_t i, j;                         //循环计数器
 for (j = 0; j < 8; j++) {              //垂直循环
        for (i = 0; i < 8; i++) {//水平循环
                dst[j * 8 + i] = (int16_t) src[j * stride + i];把读取的像素值扩展
                                                                  为16位
        }
 }
}
```

上述代码数据操作是整个图像块，比较规整 8×8，固定循环次数。

为了在 CMD 文件直接对优化后的函数进行指定内存位置存放，可定义在.text 段下再定义子段“.text:_transfer_8to16copy_asm_dm642”。需要注意的是，为了在 C 语言环境下能访问该函数，汇编函数定义时需要下画线“_”。如下例，C 语言中可以调用函数 transfer_8to16copy_asm_dm642(uint8_t *src, int16_t *dst, uint32_t stride)。

```
      .sect ".text:_transfer_8to16copy_asm_dm642"
          .global _transfer_8to16copy_asm_dm642     ; 函数名称
 _transfer_8to16copy_asm_dm642  .cproc A_o0, B_i0, A_stride     ;函数名和函数参数
          .no_mdep        ;没有存储器相关，即 A_o0和 B_i0指向两个独立的内存
          .reg  B_p_FE:B_p_DC, A_p_BA:A_p_98, B_p_76:B_p_54, A_p_32:A_p_10
          .reg  B_p_FEDC:B_p_BA98, A_p_7654:A_p_3210 ;定义的寄存器变量
          .reg  A_i1, B_o1, A_i, A_pitch
          MVK              3,          A_i                   ;循环次数，从0计数
          ADD              B_i0,       A_stride,     A_i1;指向内存图像的下一行
          ADD              A_o0,       8,            B_o1;
     SHR                 A_stride,    2,       A_pitch     ;
 loop_8to16copy: .trip 4, 4, 4    ;最小、最大迭代次数和公倍数均为4，即固定循环迭代4次
```

```
        LDDW            *A_i1++[A_pitch], B_p_FEDC:B_p_BA98  ；读取8
                        字节到寄存器对
        LDDW            *B_i0++[A_pitch], A_p_7654:A_p_3210  ;
        UNPKHU4         B_p_FEDC, B_p_FE     ；高半字的2字节送B2和B0，
                                               其余补0
        UNPKLU4         B_p_FEDC, B_p_DC     ；低半字的2字节送B2和B0，
                                               其余补0
        UNPKHU4         B_p_BA98, A_p_BA
        UNPKLU4         B_p_BA98, A_p_98
        UNPKHU4         A_p_7654, B_p_76
        UNPKLU4         A_p_7654, B_p_54
        UNPKHU4         A_p_3210, A_p_32
        UNPKLU4         A_p_3210, A_p_10
        STDW            B_p_FE:B_p_DC, *B_o1[2]   ;对齐方式8字节存储
        STDW            A_p_BA:A_p_98, *A_o0[2]   ;
        STDW            B_p_76:B_p_54, *B_o1++[4];
        STDW            A_p_32:A_p_10, *A_o0++[4];
  [A_i] BDEC            loop_8to16copy, A_i       ;条件判断，跳转到标号
        .return                                   ;函数返回
        .endproc                                  ;函数定义结束
```

在上述例程中，没有明确指明功能单元和哪个寄存器一侧，让汇编优化器来自动编排。如果读写的内存指针没有 4、8 字节对齐，则需要使用非对齐指令，但汇编优化器不能充分并行指令，程序优化的性能降低。汇编优化器对函数 transfer_8to16copy_asm_dm642 的优化结果如下。

```
_transfer_8to16copy_asm_dm642:
            ADD      .D1X     B4,A6,A3          ; |13|
||          MVC      .S2      CSR,B18
            MV       .D1X     B4,A18
||          AND      .D2      -2,B18,B4
            MVK      .D1      2,A0              ; |12|
||          MVC      .S2      B4,CSR            ; interrupts off
L1:     ; PIPED LOOP PROLOG
            MV       .D2X     A4,B16
|| [ A0]    BDEC     .S1      loop_8to16copy,A0 ; |40| (P) <0,4>
            ADD      .D2X     0x8,A4,B4         ; |14|
||          ADD      .D1      0x8,A4,A16        ; |14|
||          SHR      .S1      A6,0x2,A17        ; |15|
||          ZERO     .L1      A4
            SUB      .D2      B4,16,B17
||          SET      .S1      A4,0xf,0xf,A1     ; init prolog collapse predicate
||          LDDW     .D1T2    *A3++[A17],B9:B8; |23| (P) <0,0> ^
;** -----------------------------------------------------------------------*
loop_8to16copy:          ; PIPED LOOP KERNEL
    [!A1]   STDW     .D2T2    B7:B6,*++B17(32)  ; |35| <0,7> ^
||          UNPKHU4  .S2X     A7,B5              ; |30| <0,7> ^
```

```
||              UNPKLU4  .L1X     B8,A4                ; |29| <0,7> ^
|| [ A0]        EC       .S1  loop_8to16copy,A0  ; <1,4>
||              LDDW     .D1T1    *A18++[A17],A7:A6   ; |24| <2,1> ^
   [ A1]        MPYSU    .M1      2,A1,A1              ; <0,8>
|| [!A1]        STDW     .D1T2    B5:B4,*A16++(32)     ; |37| <0,8> ^
||              UNPKLU4  .S1      A6,A8                ; |33| <0,8> ^
|| [!A1]        STDW     .D2T1    A5:A4,*+B16(16)      ; |36| <0,8> ^
||              UNPKHU4  .S2      B9,B7                ; |26| <1,5> ^
||              UNPKLU4  .L2      B9,B6                ; |27| <1,5> ^
   [!A1]        STDW     .D2T1    A9:A8,*B16++(32)     ; |38| <0,9> ^
||              UNPKLU4  .S2X     A7,B4                ; |31| <1,6> ^
||              UNPKHU4  .S1      A6,A9                ; |32| <1,6> ^
||              UNPKHU4  .L1X     B8,A5                ; |28| <1,6> ^
||              LDDW     .D1T2    *A3++[A17],B9:B8    ; |23| <3,0> ^
;** ----------------------------------------------------------------------*
L3:      ; PIPED LOOP EPILOG
                UNPKHU4  .S2X     A7,B5                ; |30| (E) <1,7> ^
||              UNPKLU4  .S1X   B8,A4                  ; |29| (E) <1,7> ^
||              LDDW     .D1T1  *A18++[A17],A7:A6     ; |24| (E) <3,1> ^
||              STDW     .D2T2  B7:B6,*++B17(32)       ; |35| (E) <1,7> ^
                UNPKLU4  .S1    A6,A8                  ; |33| (E) <1,8> ^
||              STDW     .D2T1  A5:A4,*+B16(16)        ; |36| (E) <1,8> ^
||              UNPKHU4  .S2    B9,B7                  ; |26| (E) <2,5> ^
||              UNPKLU4  .L2    B9,B6                  ; |27| (E) <2,5> ^
||              STDW     .D1T2  B5:B4,*A16++(32)       ; |37| (E) <1,8> ^
                UNPKLU4  .S2X   A7,B4                  ; |31| (E) <2,6> ^
||              UNPKHU4  .S1    A6,A9                  ; |32| (E) <2,6> ^
||              UNPKHU4  .L1X   B8,A5                  ; |28| (E) <2,6> ^
||              STDW     .D2T1  A9:A8,*B16++(32)       ; |38| (E) <1,9> ^
                UNPKHU4  .S2X   A7,B5                  ; |30| (E) <2,7> ^
||              UNPKLU4  .S1X   B8,A4                  ; |29| (E) <2,7> ^
||              STDW     .D2T2  B7:B6,*++B17(32)       ; |35| (E) <2,7> ^
                UNPKLU4  .S1X   B8,A4                  ; |29| (E) <3,7> ^
||              STDW     .D1T2  B5:B4,*A16++(32)       ; |37| (E) <2,8> ^
||              UNPKLU4  .L1    A6,A8                  ; |33| (E) <2,8> ^
||              UNPKHU4  .S2    B9,B7                  ; |26| (E) <3,5> ^
||              UNPKLU4  .L2    B9,B6                  ; |27| (E) <3,5> ^
||              STDW     .D2T1  A5:A4,*+B16(16)        ; |36| (E) <2,8> ^
                RET      .S2    B3                     ; |42|
||              UNPKHU4  .S1    A6,A9                  ; |32| (E) <3,6> ^
||              UNPKHU4  .L1X   B8,A5                  ; |28| (E) <3,6> ^
||              UNPKLU4  .L2X   A7,B4                  ; |31| (E) <3,6> ^
||              STDW     .D2T1  A9:A8,*B16++(32)       ; |38| (E) <2,9> ^
                MVC      .S2    B18,CSR                ; interrupts on
||              UNPKLU4  .S1    A6,A8                  ; |33| (E) <3,8> ^
||              UNPKHU4  .L2X   A7,B5                  ; |30| (E) <3,7> ^
```

```
||          STDW    .D2T2   B7:B6,*++B17(32)      ; |35| (E) <3,7> ^
            STDW    .D1T2   B5:B4,*A16++(32)      ; |37| (E) <3,8> ^
            STDW    .D2T1   A5:A4,*+B16(16)       ; |36| (E) <3,8> ^
            STDW    .D2T1   A9:A8,*B16++(32)      ; |38| (E) <3,9> ^
            NOP         1
            ; BRANCH OCCURS                       ; |42|
;           .endproc
```

上述的循环核 PIPED LOOP KERNEL 的迭代间隔为 3，即 3 个 CPU 指令周期，并行了 5+6+5 条指令。应该说该软件流水还是很不错的，指令并行度较高。流水填充较短，而流水排空较长，有 1 个 NOP 指令。另外在流水前启用了关闭中断，而流水后开启中断。因为有可能有的流水程序较耗时，但 DSP 系统有其他资源需要申请中断处理。开发者可以在 CCS 集成开发环境中设置流水是否关闭中断。

2. nt_h263_intra_c

函数功能是把 Intra 宏块的 6 个块做量化，由于 Intra 块有 AC/DC 分量的区别，而两个分量的量化类型不同。一个块的 DCT 系数有 1 个 DC 值，63 个 AC 值。

```
uint32_t quant_h263_intra_c(int16_t* coeff, const int16_t* data,const uint32_t quant, const uint32_t dcscalar)
{
 const uint32_t mult = multipliers[quant];
 const uint16_t quant_m_2 = quant << 1;
 int i;
 coeff[0] = DIV_DIV(data[0], (int32_t) dcscalar);        /*量化DC分量*/
 for (i = 1; i < 64; i++)                                /*循环量化AC系数*/
 {
       int16_t acLevel = data[i];
       if (acLevel < 0){                                 /*AC系数<0*/
            acLevel = -acLevel;                          /*取反*/
            if (acLevel < quant_m_2){                    /*小于2倍量化步长*/
                 coeff[i] = 0;                           /*系数置为0*/
                 continue;
            }
            acLevel = (acLevel * mult) >> SCALEBITS;    /*量化*/
            coeff[i] = -acLevel;                         /*量化结果取反，并存储*/
       }else{                                            /*AC系数>0*/
            if (acLevel < quant_m_2){                    /*小于2倍量化步长*/
                 coeff[i] = 0;                           /*系数置为0*/
                 continue;
            }
            acLevel = (acLevel * mult) >> SCALEBITS;    /*量化*/
            coeff[i] = acLevel;                          /*存储量化结果*/
       }
 }
 return(0);
}
```

此函数是核心模块，所有的宏块都要经过该模块处理。分析来看，定循环中数据没有对齐，因为第一个数据 data[0]没有在循环内，这就无法使用对齐的数据打包指令 LDDW、STDW。多层判断无法流水。优化的方法：把第一个数据也放在循环中，循环完再单独处理第一个数据。把两层的判断使用汇编指令的条件字段，不再判断或跳转。另外模块的 C 语言函数是只处理一个块，宏块有 4 个亮度块加 2 个色度块。因此可以在汇编中一次处理 6 个块，从而减少函数调用的开销。使用线性汇编优化该函数如下所示。

```
     .sect ".text:_quant_h263_intra_asm_dm642"
          .global _quant_h263_intra_asm_dm642
  _quant_h263_intra_asm_dm642  .cproc A_coeff0, B_data0, A_quant, B_scaler_lum,A_
scaler_chr,B_multmult
          .no_mdep                                        ;没有存储器依赖性
          .reg  B_d_76:B_d_54, A_d_32:A_d_10              ;数据寄存器对data
          .reg  B_c_76:B_c_54, A_c_32:A_c_10              ;比较结果compare
          .reg  B_x_76:B_x_54, A_x_32:A_x_10              ;比较结果扩展xpnd
          .reg  B_fix_76:B_fix_54, A_fix_32:A_fix_10      ;fix
          .reg  B_p_7:B_p_6, B_p_5:B_p_4, A_p_3:A_p_2, A_p_1:A_p_0;乘积, 与1相乘
          .reg  B_p_76:B_p_54, A_p_32:A_p_10              ;数据打包packed product
          .reg  B_fix2_76:B_fix2_54, A_fix2_32:A_fix2_10
          .reg  A_i1, B_o1, A_i, A_pitch
          .reg  A_multmult,A_thth, B_thth
          .reg  B_coeff1, A_data1
          .reg  A_0101, B_0101
     MVKL              0x00010001,    A_0101
     MVKH              0x00010001,    A_0101
     MV                A_0101,        B_0101
       MV      .1X        B_multmult,    A_multmult          ; move mult
       PACK2              B_multmult,    A_multmult,     B_multmult
                                                         ;打包mult, pack mult
       MV                 B_multmult,   A_multmult       ;数值相同, twin mult

       ZERO    .1         A_thth                         ;清零, 为了比较操作, <0
       ZERO    .2         B_thth                         ;清零, 为了比较操作, <0
       MVK          47,         A_i                      ;循环计数器
       ADD     .2X     A_coeff0,     8,       B_coeff1   ;指向第4个元素
       ADD     .1X     B_data0,      8,       A_data1    ;指向第4个元素
       .mptr           B_data0,     data  + 0,   16 ;自行判定2个存储器操作是否存在冲突
       .mptr           A_data1,     data  + 8,   16 ;
       .mptr           A_coeff0,    coeff + 0,      16 ;
       .mptr           B_coeff1,    coeff + 8,      16 ;
  loop_quant_h263_intra: .trip 48, 48, 48                ;迭代48次=8×6
        LDDW           .D1T2        *A_data1++[2],     B_d_76:B_d_54
        LDDW           .D2T1        *B_data0++[2],     A_d_32:A_d_10
        CMPLT2   .2        B_d_76,     B_thth,         B_c_76   ;与0比较
        CMPLT2   .2        B_d_54,     B_thth,         B_c_54   ;
        CMPLT2   .1        A_d_32,      A_thth,        A_c_32   ;
```

```
      CMPLT2   .1      A_d_10,     A_thth,       A_c_10   ;
    XPND2   .2    B_c_76,         B_x_76                ;结果扩展
    XPND2   .2    B_c_54,         B_x_54                ;
    XPND2   .1    A_c_32,         A_x_32                ;
    XPND2   .1    A_c_10,         A_x_10                ;
    AND     .2   B_x_76,  B_0101,      B_fix_76 ;屏蔽多余的位
    AND     .2   B_x_54,  B_0101,      B_fix_54 ;
    AND     .1   A_x_32,  A_0101,      A_fix_32 ;
    AND     .1   A_x_10,  A_0101,      A_fix_10 ;
    MPY2    .2    B_d_76,     B_multmult,        B_p_7:B_p_6;两个16位相乘，
                                                          结果64位
    MPY2   .2    B_d_54,     B_multmult,       B_p_5:B_p_4 ;
    MPY2   .1    A_d_32,     A_multmult,       A_p_3:A_p_2 ;
    MPY2   .1    A_d_10,     A_multmult,       A_p_1:A_p_0 ;
    PACKH2 .2   B_p_7,       B_p_6,          B_p_76 ;相乘结果打包
    PACKH2 .2   B_p_5,       B_p_4,          B_p_54 ;
    PACKH2 .1   A_p_3,       A_p_2,          A_p_32 ;
    PACKH2 .1   A_p_1,       A_p_0,          A_p_10 ;
    ADD2   .2   B_p_76,      B_fix_76,       B_p_76 ;根据<0 、>0的结果修正最
                                                    后的值
    ADD2   .2  B_p_54,       B_fix_54,  B_p_54 ;
    ADD2   .1  A_p_32,       A_fix_32,  A_p_32 ;
    ADD2   .1  A_p_10,       A_fix_10,  A_p_10 ;
     STDW    .D2T2      B_p_76:B_p_54, *B_coeff1++[2]   ;存储量化后的结果
     STDW    .D1T1      A_p_32:A_p_10, *A_coeff0++[2]   ;
[A_i] BDEC           loop_quant_h263_intra, A_i          ;减1判断
     .return
     .endproc
```

在线性汇编中，利用了数据打包操作 PACKH2，避免了>> SCALEBITS 操作。用巧妙的与、加操作避免了小于 0 和大于 0 的两种操作的判断。一次处理了 6 个块，减少了函数的调用开销。这种利用汇编指令的特点，转化运算类型的算法优化方法是最有效的优化技术。进行线性汇编编程经常用到的关键字、伪指令如表 13-1 所示。

表 13-1　线性汇编伪指令功能描述

语法	功能描述
.call[ret=] func_name(arg1,arg2)	在线性汇编中调用一个函数
label .cproc[var1,var2[…]] .endproc	定义可以被 C 语言调用的函数 label
.no_mdep	表明存储器无相关性
.reg var1	定义寄存器变量
.return [argument]	函数返回
label .trip min,max,multiple	定义循环迭代的次数

13.4.4 使用Cache技术实现算法优化

缓冲 Cache 是将 CPU 近期访问过的数据或程序放置在片上 Cache 中，以提高 CPU 的执行速度。DM642 的 CPU 采用了两级高速缓存 Cache 结构，极大地提高了 CPU 的数据和程序的读写速度。第一级 Cache 为片上的 L1 级，以 CPU 的速度被读写。L1 又分为 L1P 程序 Cache 和 L1D 数据 Cache。其中，L1P、L1D 的大小均是 16KB。L1P 采用直接映射的组织形式，仅能作为 Cache 操作，不能映射到 C64x 存储空间，不支持写功能，以 CPU 的速度进行访问。L1D 采用双路联想组合 Cache（2-way set associative）的组织形式，仅能作为 Cache 操作，不能映射到 C64x 存储空间，不支持写功能，以 CPU 的速度进行访问。

第二级 Cache 为片上的 L2。DM642 的 L2 Cache 为 256KB，可配置为 SRAM 或 SRAM 和 Cache 两者的混合。当 L2 一部分配制为 SRAM 时，映射到 C64x 的统一地址空间内，地址从 0x00000000 开始，CPU 可以直接对 SRAM 进行存取。L2 一部分配置为 Cache 时，Cache 大小可从 32～256KB，被配置为 Cache 的 L2 SRAM 没有映射到存储器空间中，开发者不能进行存取。DM642 的 L2 Cache 可配置为 32KB、64KB、128KB 和 256KB。在视频编/解码开发中，64KB 和 128KB 更为常用。

开发者不能对 DM642 的两级 Cache 进行直接读写，Cache 有一套专门的控制机制来负责管理数据，使其有效或无效。其中 L1D、L1P 一般不需要人为干预，Cache 控制器有一套自己的管理机制，L2 的 Cache 是 Cache 优化的重点。TI 对 Cache 的控制提供了一套 API 函数：CACHE_xxx，其中经常使用的是 CACHE_clean()，它是将 Cache 的数据“踢”到 SDRAM 中，使 Cache 中的数据无效。另外 SDRAM 也可映射为 Cache，即使用 CACHE_enableCaching（CE0/CE1）函数可极大提高算法性能。

缓存 Cache 的优化思想就是把 CPU 经常被访问的资源放置在其片上，CPU 首先访问 L1D 或 L1P，若没有访问到所需的数据则访问 L2，若还没有命中则访问 SDRAM。所以除了需要把重要数据和核心代码放置片上外，还需要根据算法的流程有效地安排数据或代码的内存存放位置，让前后彼此相关性很强的数据和代码紧排在一起，以使 CPU 的 L1D 和 L1P 没有产生或尽可能少地不命中（Miss）。

Cache 配置一般在程序的开始进行，如下例就是在没有使用 BIOS 的情况下，调用视频算法库以测试其效率和验证算法的正确性。

```
#if defined(CHIP_DM642)
 CSL_init();                                          //初始化CSL库
 CACHE_clean(CACHE_L2ALL, 0, 0);                      //L2的Cache清空
 CACHE_setL2Mode(CACHE_128KCACHE);                    //L2设置为128KB Cache
 CACHE_enableCaching(CACHE_EMIFA_CE00);               //SDRAM0使能Cache
 CACHE_enableCaching(CACHE_EMIFA_CE01);               //SDRAM1 使能Cache
 DAT_open(DAT_CHAANY, DAT_PRI_LOW, DAT_OPEN_2D);      //使用CSL打开DMA通道
 CACHE_setPriL2Req(CACHE_L2PRIHIGH);                  //在CCFG寄存器设置L2的P优先级
#endif
```

13.4.5 使用乒乓式EDMA实现算法优化

由于DM642外接的扩展存储器是SDRAM，而SDRAM工作频率为133MHz，这与600MB的CPU速度不匹配。即CPU访问SDRAM中的数据需要插入多个nop指令以确保存取成功，这就极大地限制了CPU的高速运行，从而算法不能实时地、高效地处理数据。为了提高CPU的Cache命中率，用户需要尽可能地把CPU要访问的数据放置在片上L1或L2中，开发者不能直接访问L1，L2可以配置为SRAM和Cache组合。乒乓式缓冲即使用两个缓冲区，一个当前程序使用，一个后台DMA传输。使用DMA的双缓冲方式，把SDRAM图像或空间映射到L2的SRAM，保证CPU要处理的图像数据或空间是在L2中。在CPU的后台使用DMA方式完成图像的L2到SDRAM或SDRAM到L2的传输。

DM642的DMA采用了增强型DMA即EDMA，通道更多、传输方式更灵活，丰富了图像处理传输的DMA技术。视频编/解码算法的EDMA优化是优化策略的系统层，即更改数据流的流向、数据结构。对视频编码算法来说，乒乓式EDMA实现算法优化是把当前编码图像和重建参考图像的数据映射到L2的SRAM，使用DMA双缓冲复制这些图像；对视频解码算法来说，把当前图像空间和参考图像空间映射到L2的SRAM，使用DMA双缓冲复制这些空间。

DSP的EDMA的使用有几种方式。一是使用DAT_XX类CSL库函数，TI已经封装好了DMA的操作过程：DAT_open()、DAT_copy()、DAT_wait()、DAT_close()；二是使用EDMA_XX函数，TI也提供了可直接调用的API函数。

第一种方法编程简单，更为常用。另外在打开DMA的时候，要特别注意DMA通道的优先级，因为工程的其他模块如音频的编码、解码也需要DMA。优先级配置不合适，会影响系统的正常运行。在使用EDMA搬移数据时，需根据算法的处理过程合理地安排传输时间，以便有效利用EDMA后台工作的特点。如在缓存参考帧的数据时，当要开始处理这一行宏块时，先开启下次需要的宏块数据的传输。为了保持数据的一致性，不可对一段SRAM同时进行读和写操作。

下面以MPEG-4视频编码算法中的I帧编码函数FrameCodeI为例，展示如何使用双缓冲的DMA技术。

```
int FrameCodeI(Encoder * pEnc, Bitstream * bs)
{
 int bits = BitstreamPos(bs);
 uint32_t mb_width     = pEnc->mbParam.mb_width;        //图像宏块宽度
 uint32_t mb_height    = pEnc->mbParam.mb_height;       //图像宏块高度
 uint32_t edged_width = pEnc->mbParam.edged_width;     //图像扩展后的宽度=32+
                                                         width+32
 uint32_t size_MB_line   = edged_width<<4;             //一行亮度宏块的大小
 uint32_t size_MB_line2  = edged_width<<2;             //一行色度宏块的大小
 uint16_t x, y;
 IMAGE orgCurImage = pEnc->current->image;             //当前编码图像空间
 IMAGE bufImage0,bufImage1;                            //图像缓存空间
```

```
    uint32_t    transCurY0,transCurU0,transCurV0;              //DMA通道号
    uint32_t    transCurY1,transCurU1,transCurV1;              //DMA通道号
    pEnc->current->coding_type = I_VOP;                        //当前帧编码类型为I帧
    pEnc->current->mbs[0].quant = pEnc->current->quant;        //获取量化步长
    SetMacroblockQuants(&pEnc->mbParam, pEnc->current);        //设置图像宏块的量化
    BitstreamWriteVolHeader(bs, &pEnc->mbParam, pEnc->current);  //写VOL头信息
    set_timecodes(pEnc->current,pEnc->reference,pEnc->mbParam.fbase);
    BitstreamPad(bs);                                            //写VOP头信息
    BitstreamWriteVopHeader(bs, &pEnc->mbParam, pEnc->current, 1, pEnc->current->
mbs[0].quant);
    //初始化编码统计信息
    pEnc->current->sStat.iTextBits = 0;
    pEnc->current->sStat.kblks = mb_width * mb_height;
    pEnc->current->sStat.mblks = pEnc->current->sStat.ublks = 0;
   #if 0 //没有使用DMA缓冲图像，CPU访问的图像在SDRAM内
    for (y = 0; y < mb_height; y++){
            /*编码一行宏块图像，下面的图像编码即该部分代码实现*/
                for (x = 0; x < mb_width; x++)
                {
                MACROBLOCK *pMB = &pEnc->current->mbs[x + y * pEnc->mbParam.mb_width];
                CodeIntraMB(pEnc, pMB);
                    //图像Intra宏块DCT变换、量化
                MBTransQuantIntra(&pEnc->mbParam,pEnc->current,pMB,x,y,dct_coeff,
qnt_coeff);
                //I帧图像DCT系数预测
                    MBPrediction(pEnc->current, x, y, pEnc->mbParam.mb_width, qnt_
coeff);
                    //系数VLC熵编码
                MBCoding(pEnc->current, pMB, qnt_coeff, bs, &pEnc->current->sStat);
        }
        }
   #else //使用DMA双缓冲编码图像
    bufImage0.y = buf_cur_fix_y0;    //指向L2上的实际亮度编码空间
    bufImage0.u = buf_cur_fix_u0;    //指向L2上的实际色度编码空间
    bufImage0.v = buf_cur_fix_v0;    //指向L2上的实际色度编码空间
    bufImage1.y = buf_cur_fix_y1;
    bufImage1.u = buf_cur_fix_u1;
    bufImage1.v = buf_cur_fix_v1;
       //0行宏块，y=0
    //从SDRAM复制0/1行图像亮度/色度宏块数据到L2
    y=0;
    transCurY0=DAT_copy(orgCurImage.y+(y+0)*size_MB_line
-EDGE_SIZE  ,buf_cur_y0,size_MB_line);
    transCurU0=DAT_copy(orgCurImage.u+(y+0)*size_MB_line2-EDGE_SIZE2,buf_cur_u0,
size_MB_line2);
    transCurV0=DAT_copy(orgCurImage.v+(y+0)*size_MB_line2-EDGE_SIZE2,buf_cur_v0,
```

```
size_MB_line2);
    transCurY1=DAT_copy(orgCurImage.y+(y+1)*size_MB_line-EDGE_SIZE  ,buf_cur_y1,
size_MB_line);
    transCurU1=DAT_copy(orgCurImage.u+(y+1)*size_MB_line2-EDGE_SIZE2,buf_cur_u1,
size_MB_line2);
    transCurV1=DAT_copy(orgCurImage.v+(y+1)*size_MB_line2-EDGE_SIZE2,buf_cur_v1,
size_MB_line2);
    pEnc->current->image = bufImage0;        //编码函数中的当前图像空间映射到L2的
bufImage0
    DAT_wait(transCurY0);                    //等待确保数据DMA传送完毕
    DAT_wait(transCurU0);
    DAT_wait(transCurV0);
    //对第0行图像以宏块为单位循环编码
    //从L2更新到SDRAM
    transCurY0=DAT_copy(buf_cur_y0,orgCurImage.y+(y+0)*size_MB_line-EDGE_SIZE  ,
size_MB_line);
    transCurU0=DAT_copy(buf_cur_u0,orgCurImage.u+(y+0)*size_MB_line2-EDGE_SIZE2,
size_MB_line2);
    transCurV0=DAT_copy(buf_cur_v0,orgCurImage.v+(y+0)*size_MB_line2-EDGE_SIZE2,
size_MB_line2);
    //y==1
    y=1;
    pEnc->current->image = bufImage1;           //编码函数中的当前图像空间映射到L2的
                                                  bufImage1
    DAT_wait(transCurY1);                       //等待确保数据DMA传送完毕
    DAT_wait(transCurU1);
    DAT_wait(transCurV1);
    //对第1行图像以宏块为单位循环编码
       //..............................
       //等待传送完毕
    DAT_wait(transCurY0);                       //等待确保数据DMA传送完毕
    DAT_wait(transCurU0);
    DAT_wait(transCurV0);
       //从SDRAM复制第2行图像亮度/色度宏块数据到L2
    transCurY0=DAT_copy(orgCurImage.y+(y+1)*size_MB_line-EDGE_SIZE,buf_cur_y0,
size_MB_line);
    transCurU0=DAT_copy(orgCurImage.u+(y+1)*size_MB_line2-EDGE_SIZE2,buf_cur_u0,
size_MB_line2);
    transCurV0=DAT_copy(orgCurImage.v+(y+1)*size_MB_line2-EDGE_SIZE2,buf_cur_v0,
size_MB_line2);
    //从L2更新到SDRAM
    transCurY1=DAT_copy(buf_cur_y1,orgCurImage.y+(y+0)*size_MB_line-EDGE_SIZE  ,
size_MB_line);
    transCurU1=DAT_copy(buf_cur_u1,orgCurImage.u+(y+0)*size_MB_line2-EDGE_SIZE2,
size_MB_line2);
    transCurV1=DAT_copy(buf_cur_v1,orgCurImage.v+(y+0)*size_MB_line2-EDGE_SIZE2,
```

```
size_MB_line2);
    //loop y
    for (y=2; y < mb_height-2; y +=2)          //注意y是从2开始，且每次递增2!
    {
    // y
    pEnc->current->image = bufImage0;
    DAT_wait(transCurY0);                      //等待确保数据DMA传送完毕
    DAT_wait(transCurU0);
    DAT_wait(transCurV0);
       //对第y行图像以宏块为单位循环编码
       //……
       //等待传送完毕
    DAT_wait(transCurY1);                      //等待确保数据DMA传送完毕
    DAT_wait(transCurU1);
    DAT_wait(transCurV1);
       //从SDRAM复制第y+1行图像亮度/色度宏块数据到L2
    transCurY1=DAT_copy(orgCurImage.y+(y+1)*size_MB_line-EDGE_SIZE,buf_cur_y1,
size_MB_line);
    transCurU1=DAT_copy(orgCurImage.u+(y+1)*size_MB_line2-EDGE_SIZE2,
buf_cur_u1,size_MB_line2);
    transCurV1=DAT_copy(orgCurImage.v+(y+1)*size_MB_line2-EDGE_SIZE2,
buf_cur_v1,size_MB_line2);
    //从L2更新到SDRAM
    transCurY0=DAT_copy(buf_cur_y0,orgCurImage.y+(y+0)*size_MB_line-EDGE_SIZE  ,
size_MB_line);
    transCurU0=DAT_copy(buf_cur_u0,orgCurImage.u+(y+0)*size_MB_line2-EDGE_SIZE2,
size_MB_line2);
    transCurV0=DAT_copy(buf_cur_v0,orgCurImage.v+(y+0)*size_MB_line2-EDGE_SIZE2,
size_MB_line2);
    //y+1
    pEnc->current->image = bufImage1;
    DAT_wait(transCurY1);                      //等待确保数据DMA传送完毕
    DAT_wait(transCurU1);
    DAT_wait(transCurV1);
       //对第y+1行图像以宏块为单位循环编码
       //……
       //等待传送完毕
    DAT_wait(transCurY0);                      //等待确保数据DMA传送完毕
    DAT_wait(transCurU0);
    DAT_wait(transCurV0);
       //从SDRAM复制第y+1+1行图像亮度/色度宏块数据到L2
   transCurY0=DAT_copy(orgCurImage.y+(y+1+1)*size_MB_line-EDGE_SIZE,buf_cur_y0,
size_MB_line);
    transCurU0=DAT_copy(orgCurImage.u+(y+1+1)*size_MB_line2-EDGE_SIZE2,buf_cur_
u0,size_MB_line2);
    transCurV0=DAT_copy(orgCurImage.v+(y+1+1)*size_MB_line2-EDGE_SIZE2,buf_cur_
```

```
v0,size_MB_line2);
      //从L2更新到SDRAM
     transCurY1=DAT_copy(buf_cur_y1,orgCurImage.y+(y+1+0)*size_MB_line-EDGE_SIZE ,
size_MB_line);
     transCurU1=DAT_copy(buf_cur_u1,orgCurImage.u+(y+1+0)*size_MB_line2-EDGE_
SIZE2, size_MB_line2);
     transCurV1=DAT_copy(buf_cur_v1,orgCurImage.v+(y+1+0)*size_MB_line2-EDGE_
SIZE2, size_MB_line2);
     }
    //y==mb_height-2
     y=mb_height-2;
     pEnc->current->image = bufImage0;
     DAT_wait(transCurY0);                    //等待确保数据DMA传送完毕
     DAT_wait(transCurU0);
     DAT_wait(transCurV0);
        //对第mb_height-2行图像以宏块为单位循环编码
        //等待传送完毕
     DAT_wait(transCurY1);                    //等待确保数据DMA传送完毕
     DAT_wait(transCurU1);
     DAT_wait(transCurV1);
        //从SDRAM复制第mb_height-1行图像亮度/色度宏块数据到L2
     transCurY1=DAT_copy(orgCurImage.y+(y+1)*size_MB_line -EDGE_SIZE , buf_cur_y1,
size_MB_line);
    transCurU1=DAT_copy(orgCurImage.u+(y+1)*size_MB_line2 -EDGE_SIZE2 ,buf_cur_u1,
size_MB_line2);
    transCurV1=DAT_copy(orgCurImage.v+(y+1)*size_MB_line2 -EDGE_SIZE2 ,buf_cur_v1,
size_MB_line2);
        //从L2更新到SDRAM
     transCurY0=DAT_copy(buf_cur_y0,orgCurImage.y+(y+0)*size_MB_line-EDGE_SIZE,
size_MB_line);
    transCurU0=DAT_copy(buf_cur_u0,orgCurImage.u+(y+0)*size_MB_line2 -EDGE_SIZE2 ,
size_MB_line2);
    transCurV0= DAT_copy(buf_cur_v0,orgCurImage.v+(y+0)*size_MB_line2-EDGE_SIZE2 ,
size_MB_line2);
    //y==mb_height-1
    y=mb_height-1;
     pEnc->current->image = bufImage1;
     DAT_wait(transCurY1);                //等待确保数据DMA传送完毕
     DAT_wait(transCurU1);
     DAT_wait(transCurV1);
        //对第 mb_height-1行图像以宏块为单位循环编码
        //..............................
        //从L2更新到SDRAM
    transCurY1=DAT_copy(buf_cur_y1,orgCurImage.y+(y+0)*size_MB_line   -EDGE_SIZE,
size_MB_line);
    transCurU1=DAT_copy(buf_cur_u1,orgCurImage.u+(y+0)*size_MB_line2-EDGE_SIZE2 ,
```

```
size_MB_line2);
    transCurV1=DAT_copy(buf_cur_v1,orgCurImage.v+(y+0)*size_MB_line2-EDGE_SIZE2,
size_MB_line2);
        //等待所有DMA通道传送完毕
     DAT_wait(transCurY0);
     DAT_wait(transCurU0);
     DAT_wait(transCurV0);
     DAT_wait(transCurY1);
     DAT_wait(transCurU1);
     DAT_wait(transCurV1);
     pEnc->current->image = orgCurImage;              //回复当前编码图像指针空间
    #endif
     BitstreamPadAlways(bs);                          //填充以字节对齐
     pEnc->current->length = (BitstreamPos(bs) - bits) / 8; //编码后的码流长度
     pEnc->mbParam.m_fcode = 1;                       //运动估计搜索窗口大小因子
     pEnc->current->is_edged = 0;                     //图像没有做扩展标识
     return 1;                                        //intra编码标识
    }
```

分析上述例程可知，代码明显变大，增加了 3 倍多。在图像编码后，还需要把 L2 的图像数据更新到 SDRAM 内，因为当前图像已经修改为重建图像，为保证解码时参考帧的一致性，需要把处理后的数据再更新到片外的 SDRAM。双缓冲的空间是一行宏块图像的左边和右边都扩展了 32 个像素，但在 DMA 复制时，考虑到传送速度，没有使用二维 DMA，而是使用了一维 DMA，图像的扩展边也参与传输。尽管传输了多余的数据，但是一维 DMA 的传送效率高于二维 DMA。

13.5 DM642优化视频算法常见问题

DM642 的视频算法多数是以库的形式发布的，应用程序调用编码库。下面以一个具体的案例开发为对象，展示 DM642 的算法库工程和应用工程建立过程，给出特别要注意的设计要点。

STEP 01 建立库工程，在 CCS 开发环境中选择“Project”→“New”命令，弹出如图 13-24 所示的对话框，库工程名为“enc_lib”。

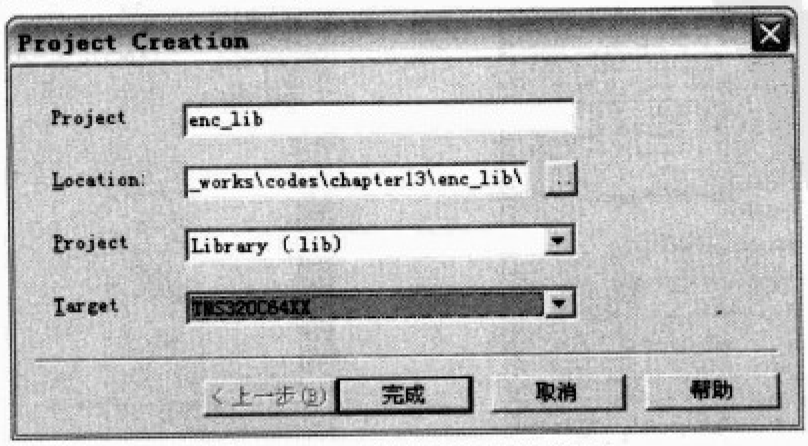


图 13-24　创建库工程向导

STEP 02 建立源文件，并编辑如下源代码 enc.c 和 enc.h，保存到当前工程目录下并把 enc.c 添加到工程中。

```
//enc.c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "enc.h"
int func_add(int a, int b) {          //函数func_add
 return a+b;                          //函数功能实现加法
}
int func_sub(int a, int b) {          //函数func_sub
 return a-b;                          //函数功能实现减法
}
```

enc.h 头文件如下：

```
//enc.h
#ifndef _ENC_LIB_HEAD
#define _ENC_LIB_HEAD 1
int func_add(int a, int b);
int func_sub(int a, int b);
#endif
```

STEP 03 修改工程编译选项，因为一般算法库都比较复杂，文件之间的函数和数组会交错访问，为保证 CPU 访问存储器地址空间不受 32KB 的限制，将内存模型（Memory Models）更改为“Far Calls & Data”，如图 13-25 所示。为确保 CPU 能访问的存储器空间不受限制，对于大型的全局静态数组，需要将其设置为 Far 类型，这一点要特别注意！如果算法库简单、小型程序，内存模型可以设置为 Near Calls & Data。这两种编译设置对编译器的编译效率影响不大。

STEP 04 定义 DSP 芯片类型。当在库中使用 CSL（Chip Surpport Library）库函数时，必须设置芯片类型，因为不同的芯片有不同的 CSL。这里使用 DM642，所以定义字符 CHIP_DM642，如图 13-26 所示。

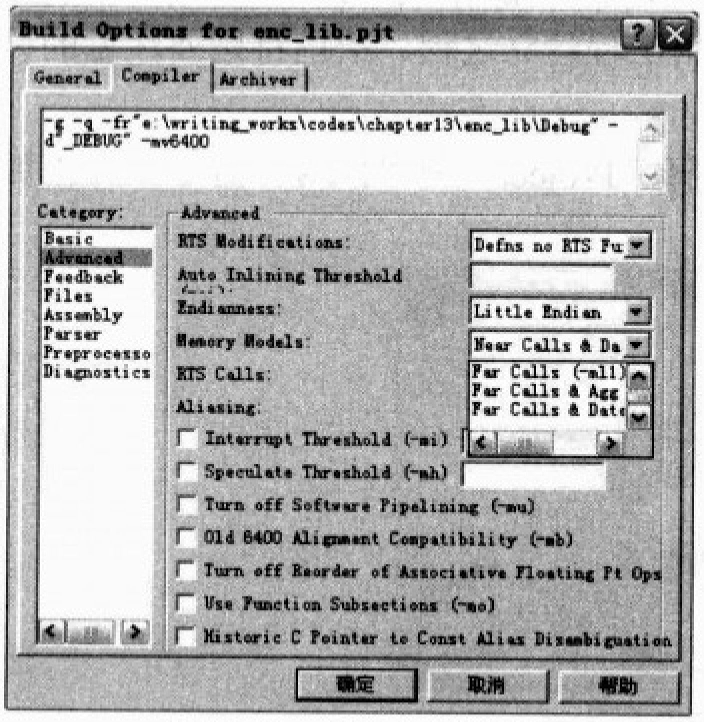


图 13-25　修改内存模型

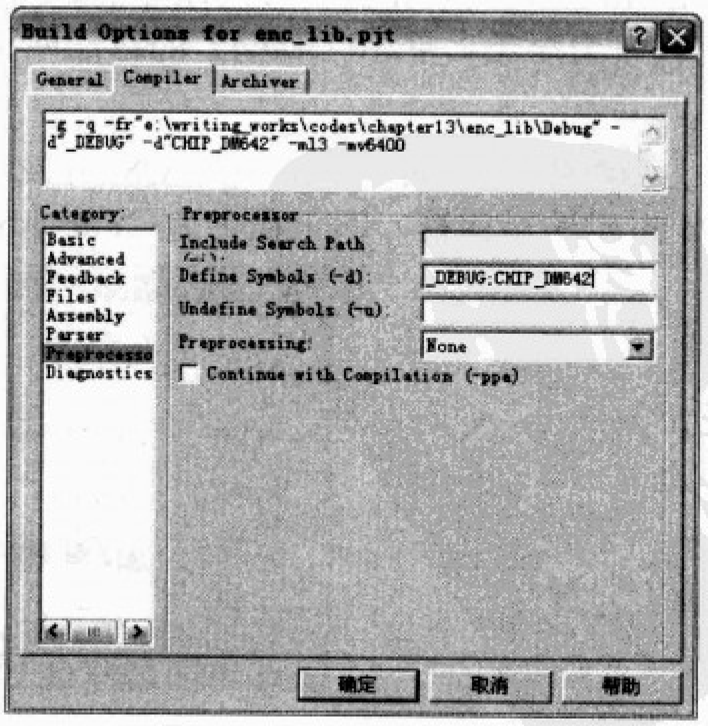


图 13-26　定义芯片类型

STEP 05 编译库工程，确定没有编译错误。若有错误就检查语句等，编译结果如图 13-27 所示。

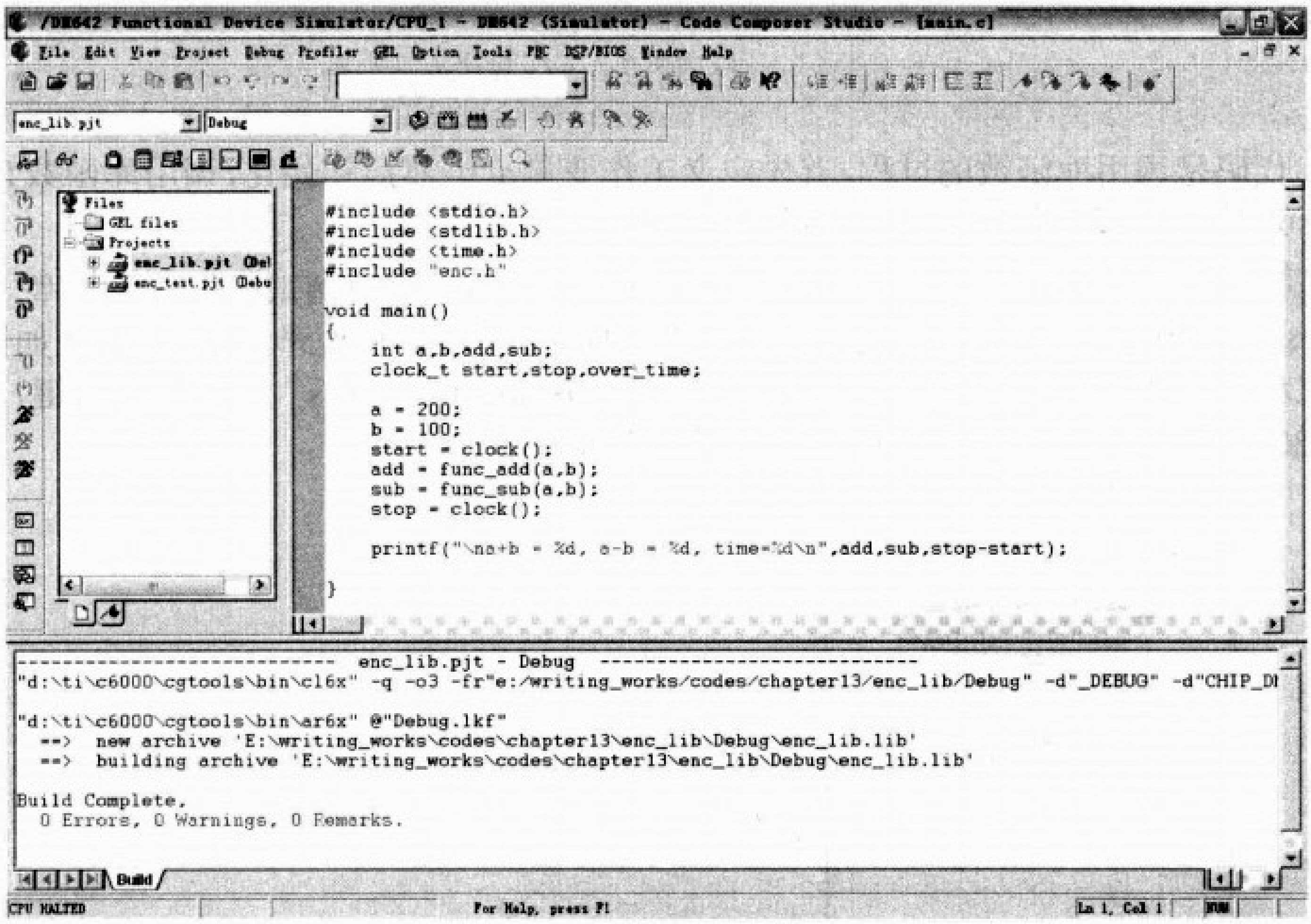


图 13-27　编译结果

STEP 06 创建应用程序工程，调用上述的算法库 enc_lib.lib。新建应用工程命名为“enc_test”，如图 13-28 所示。

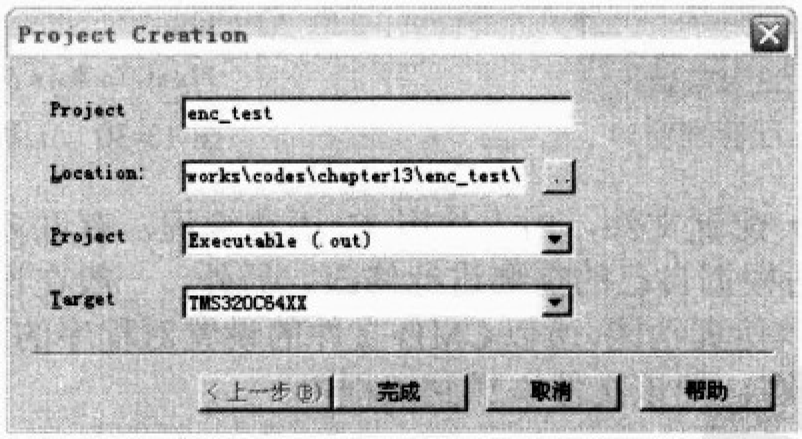


图 13-28　创建应用程序工程

STEP 07 建立源文件 main.c。把库的头文件 enc.h 复制到当前应用工程 enc_test 的目录下。

```
#include <stdio.h>
#include <stdlib.h>
#include "enc.h"
void main()
{
 int a,b,add,sub;                //工作变量
 a = 200;                        //变量初始化
```

```
    b = 100;                        //变量初始化
    add = func_add(a,b);            //调用库函数
    sub = func_sub(a,b);            //调用库函数
    printf("a+b = %d, a-b = %d\n",add,sub);  //输出运行结果
}
```

上述代码是调用库函数的过程，首先定义工作变量a、b。然后初始化，调用库函数func_add和func_sub完成计算，最后打印输出结果。

STEP 08 建立工程依赖性，使应用工程依赖于库工程，这对调试库很方便。库一旦做了修改，在编译应用工程时，编译器会自动首先编译库工程，然后再编译应用工程。在如图13-29所示的工程管理器窗口中，对当前应用工程的“Dependent Projects”单击鼠标右键，可看到如图13-30所示的快捷菜单，选择“Add Dependent Projects”命令。

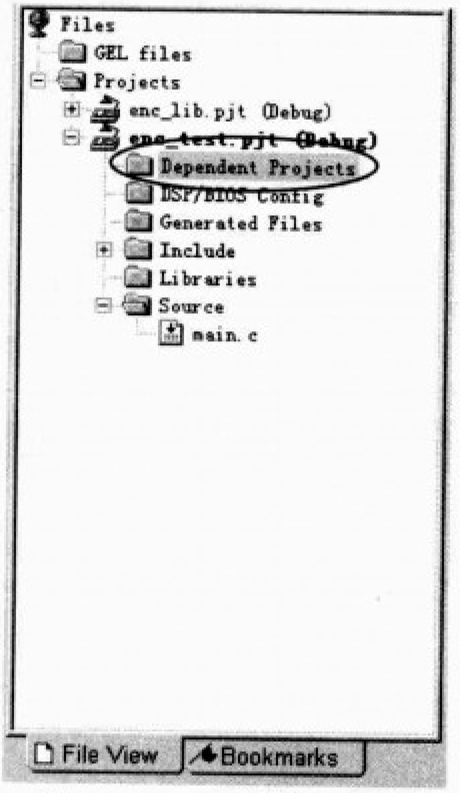


图13-29 工程管理窗口

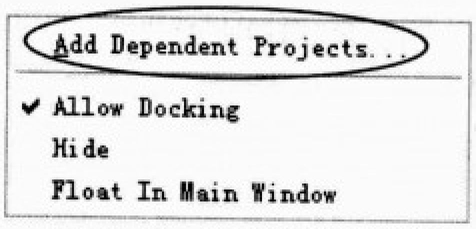


图13-30 工程依赖性菜单

STEP 09 建立CMD配置文件，由于使用C语言编程，有许多段需要定义。一般是从TI的例程复制过来，然后根据自己的需要进行修改。另外，一般在设计、优化算法库的初期是使用CCS的simulator软仿真功能，所以CMD文件的设置对程序的运行结果没有实质影响。如下是lnk.cmd文件的内容，把该文件添加到工程中。

```
MEMORY
{
 ISRAM: o = 0x00001000   l = 0x00020000
 SDRAM: o = 0x80000000   l = 0x02000000
}
SECTIONS
{
        .text           > ISRAM   /*映射到片上空间*/
        /*以下段映射到片外的SDRAM*/
        .stack          > SDRAM
        .bss            > SDRAM
        .cinit          > SDRAM
```

```
        .cio            > SDRAM
        .const          > SDRAM
        .data           > SDRAM
        .switch         > SDRAM
        .sysmem         > SDRAM
        .far            > SDRAM
}
-l cslDM642.lib
-l rts6400.lib
-l img64x.lib
```

STEP 10 库 img64x.lib 是 TI 提供的图像处理库，至此编译程序，编译器会提示“can't find input file 'img64x.lib'”。设置该库的路径“$(Install_dir)/c6400/imglib/lib”，如图 13-31 所示。如果要设置多个库的路径，则之间用分号“；”隔开。img64x.lib 库包含了图像处理的经典算法库，都经过汇编优化，效率特别高。

STEP 11 设置堆栈大小，为 malloc 等函数申请空间，为堆栈设置空间。如果不设置堆栈，则 CCS 会用默认的设置大小。“Heap Size”是为了动态申请内存函数预留空间的，用户使用 malloc 函数申请的内存就在 Heap 内。“Stack Size”是在函数调用时参数压栈，函数内部的变量和静态数组都在 Stack 中申请。在函数调用太深、层次太多而函数内部又有较大的静态数组时，由于 Stack Size 设置不合适，尽管编译不会报错，但是在程序执行时，程序无法正确返回，会经常跑飞。所以 Stack Size 应设置得足够大，设置的 Heap 和 Stack 如图 13-32 所示。

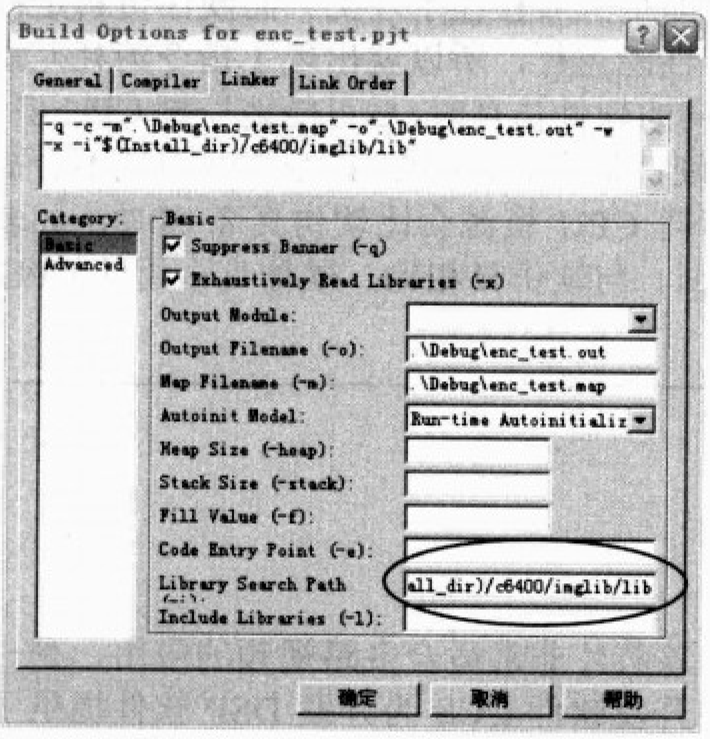


图 13-31　设置库搜索路径

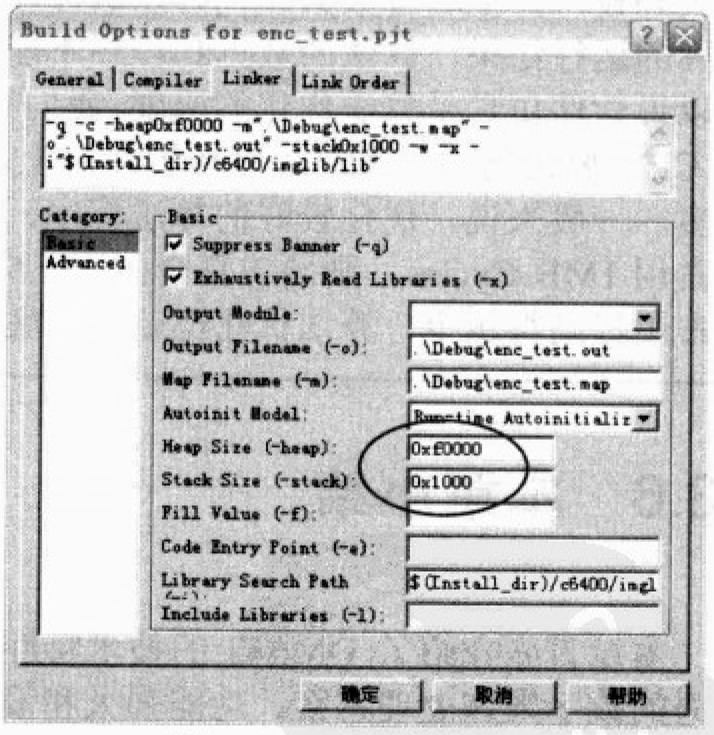


图 13-32　设置堆栈大小

STEP 12 编译应用程序工程，选择“Project”→“Build”命令，生成可执行文件*.out。若有其他错误，检查并修改。

STEP 13 下载可执行文件 enc_test.out。选择“File”→“Load Program”命令，打开“enc_test.out”文件，装载可执行文件如图 13-33 所示。

STEP 14 按“F5”键运行程序，查看输出结果窗口如图 13-34 所示。

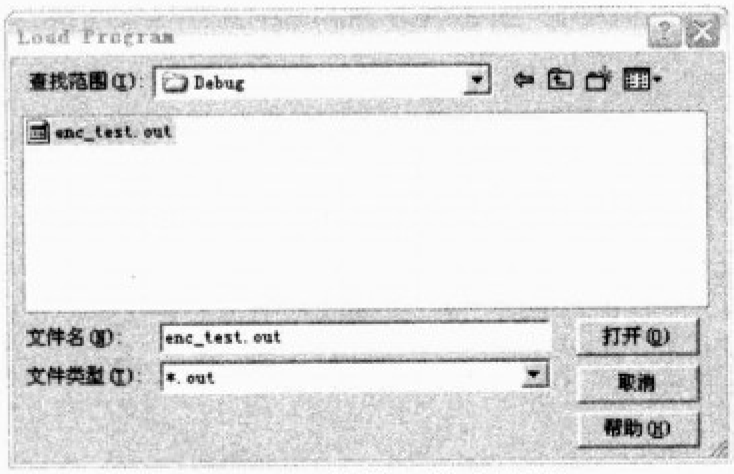


图 13-33　装载可执行文件

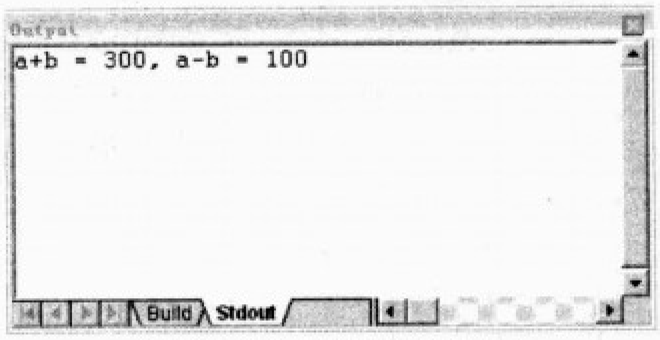


图 13-34　输出结果窗口

以上展示了典型的算法库和应用工程的创建过程，在此基础上可以丰富算法库的功能，增加函数输出接口。每次修改库的源代码、库工程编译选项如-o3 等，编译器编译应用工程时都会先编译库工程，并把结果库链接到应用工程中。

在算法设计、优化的后期就需要把算法程序应用到硬件中，即硬仿真 Emulator 如评估板 EVM。程序在 DSP 芯片中运行，这能够真实地展现算法的运行效率。TI 的硬仿真支持对文件的读写，所以在对视频编/解码算法库优化时，能够对标准的视频测试序列文件做测试和实验，以保证算法效率的可比性。

开发者在做算法的软仿真和硬仿真时，会注意到相同的程序代码做测试的 CPU 指令消耗差别较大。软件仿真 Simulator 的原理是 CCS 在计算机的内存开辟一个空间来模拟 CPU 及有关外设，并把所有数据和代码以最佳性能运行，即全部放置在了片上。而硬件仿真是程序实际的运行结果，算法的数据和程序不能都以最佳性能运行，所以就出现了指令消耗差别。可以说软件仿真是算法优化的极限，我们只有在初期的软件仿真时，尽可能地挖掘 CPU 资源，把 CPU 要访问的数据和程序放置在片上。才会在硬件仿真时，CPU 指令资源消耗不会提高太多。一般来说，优化较好的算法程序，硬仿真消耗 CPU 资源会比软仿真高 50%，如软仿真占用 1MB Cycles，则硬仿真会占用 1.5MB Cycles。与软仿真相比，硬仿真占用的 CPU 资源占用提高得越少，算法优化得就越彻底。

13.6　本章小结

本章首先介绍了 DM642 的技术特性和指标，该芯片非常适合多路视频的应用，是一款优秀的数字媒体处理方案。功能强大和设计优化的开发软件 CCS 为开发 DSP 软件提供了支持。接着介绍了 CCS 的编译优化工具、实时操作系统 DSP/BIOS 等，这些工具为算法优化和系统软件设计提供了支撑。通用的 C 语言编程、高效的 C 编译器、简单易懂的线性汇编等为软件开发和优化提供了保障。然后详细地介绍了 DM642 的视频开发软件编程技术：使用软件流水、线性汇编、Cache 技术和 EDMA 等联合使用，实现视频算法的深度优化。最后以一个具体的案例，对库、应用工程的建立到各自的编译选项设置等做了展示和分析。

第14章

XviD CODEC 实现 MPEG-4 编/解码

视频编/解码是视频应用的核心技术，如可视电话、视频监控、DVD、视频点播（VOD）等。ISO 制定的 MPEG-1 编/解码标准在 VCD 中应用，MPEG-2 在 DVD、数字电视等高质量图像中应用。为满足多媒体技术的发展及人们更多的技术需求，ISO 于 1998 年制定了 MPEG-4 音/视频编/解码标准，其码流更低、档级更丰富，如在兼容传统视频编码的基础上增加了基于对象的编码，交互性更强。视频监控 DVR 应用中以 MPEG-4 标准为主流，这些应用中有专用芯片（ASIC）、DSP 可编程芯片、软件编程 CPU 实现等技术方案。MPEG-4 的工程实现从最早的微软，到后来的 DivX、XviD。目前大家移植开发应用的都是 XviD，它是公认的最好的开源 MPEG-4 算法工程。

本章重点

- MPEG-4 编/解码概述
- XviD CODEC 编/解码分析
- 运行 XviD CODEC 系统
- 系统效果展示

PDG

14.1 MPEG-4编/解码概述

视频编/解码已经是一个非常炽热的行业，有许多公司、企业在基于各种平台 ASIC、DSP、FPGA、ARM 或普通 CPU，如 Intel/AMD 等，实现视频编/解码应用。视频监控、视频会议、网络视频、数字硬盘录像机、MP4 等影音设备、影音文件都无一不使用视频编码和解码。视频编/解码是一个有广大应用前景的行业，未来的 3G 移动通信就可以实现彼此视频图像的收发。数字电视中的 MPEG-2 标准、主流多媒体压缩板卡、嵌入式视频编码卡、视频会议等采用了 MPEG-4、H.263 等，以及国产的 AVS、国际的 H.264/AVC、微软的 WMV 9 等在视频应用中视频编/解码算法占据了核心地位。

XviD 是开源的 MPEG-4 视频编/解码 CODEC，标准 C 语言开发、部分核心函数采用了 MMX/SSE/SSE2 媒体汇编指令优化。XviD 实现了 MPEG-4 标准中的 ASP（Advanced Simple Profile），编/解码效率能够在双核 Intel CPU 1.6GB、1GB 内存配置的计算机上，实时运行 4 路 D1 的视频编码。下面就介绍 XviD 实现 MPEG-4 编码和解码算法的过程。

14.1.1 基于对象的MPEG-4视频编码

MPEG-4（ISO/IEC14496）算法是 ISO 的运动图像专家组 MPEG（Moving Pictures Expert Group）于 1998 年发布的。它设计的初衷是第二代图像编码标准，即对象编码。不过，该算法并不包含对象的分割方法，只是提供了对分割对象后的编码方法。但是，对象分割目前还是一个难点，还没有较好的突破。所以，目前的 MPEG-4 视频算法应用基本都是基于像素的传统视频编码，即混合编码技术，编码对象不是目标对象而是图像宏块，但算法中仍然有视频编码对象的概念 VOP，是把整帧图像作为一个对象进行编码的。

14.1.2 XviD格式文件播放

目前，网络上 DivX 和 XviD 格式的电影、影音文件并存。系统只安装 DivX 5 解码器不能播放 XviD 格式的文件。而只安装 XviD，则可以顺利播放 DivX 5 格式的文件。只是在播放 DivX 5 文件的时候，速度不是很令人满意。XviD 在播放 DivX 5 的文件还不是很完善。虽然 XviD 目前来讲与 DivX 5 相比，优势不是太明显。但是大家很看好 XviD，首先它的源代码绝对公开，这就使得有更多的人投入到 XviD 的研发之中。另外，由于完全重写 DivX 的源代码，新的 XviD 去除了 DivX 的 Bug；目前 XviD 的开发人员有很多都是当初 DivX 的研发人员，对 DivX 的错误了解得非常清楚，重写之后 XviD 的优势可见一斑。另外，DivX 4、DivX 5 虽然版本不断更新，但是功能并没有提高多少，优势不明显。流行的视频 CODEC 都支持 XviD：Transcode、Mencoder、Mplayer 等。

14.2 XviD CODEC编/解码分析

XviD Video CODEC 实现的是 MPEG-4 的 SP（Simple Profile）和 ASP（Advanced Simple Profile）标准，CODEC 包括视频编码算法和视频解码算法。XviD CODEC 于 2001 年创立，是 MPEG-4 视频编码算法的实现。早期的版本（0.9.x）实现了 MPEG-4 SP 版本的编码和解码，XviD 1.0 版本及其子版本实现了 MPEG-4 ASP 视频压缩，包括所有高级编码工具，如码流控制、B 帧编码、1/4 像素运动补偿和全局运动补偿 GMC 等。即将开始研发的 XviD 2.0 增加了对 MPEG-4/AVC（Advanced Video Coding）等更高档级的编/解码支持，编码压缩性能相比早期版本将大幅提升。

XviD CODEC 编/解码算法的工作流程基本都包含 3 个过程：初始化 CODEC、使用 CODEC、销毁 CODEC。初始化 CODEC 过程完成 CODEC 的创建句柄、分配必须的内存空间；循环使用 CODEC 完成视频图像的编码、MPEG-4 码流的解码；销毁 CODEC 完成初始化的反工作。XviD CODEC 使用标准 C 语言编程，同时 CODEC 的核心模块图像块类型转换复制、DCT 变换、系数量化、SAD 计算、运动补偿 MC、CBP 计算等都实现了多个 CPU：X86 的 32/64 位、ia64 等的汇编优化。

14.2.1 MPEG-4编/解码设计与剖析

XviD CODEC 以动态库和静态库的形式供应用程序使用。这里以静态库为对象，展现视频编码和解码的设计过程。通过代码分析深入掌握视频编/解码的工作原理。XviD 的 MPEG-4 算法的视频编码和解码在同一个工程中。视频编码使用一个函数，根据传入的参数决定初始化、编码和销毁编码器，视频解码函数调用与之类似。

1. MPEG-4 视频编码

XviD 的 MPEG-4 视频编码支持 ASP@L5 档级，并且绝大部分的视频应用不超过 720 像素×576 像素的分辨率，尽管 XviD 最高可支持 4096 像素×4096 像素分辨率的视频编码。而实际上在视频监控应用中，CIF（352 像素×288 像素）分辨率更常用。另外，尽管 XviD CODEC 支持高档应用：B 帧编码、1/4 精度像素和全局运动补偿 GMC。但是在使用如 DSP 硬件编码时，这些功能是简化了的，即实现 MPEG-4 的 SP 档级。而在 PC 上开发 XviD MPEG-4 算法时，这些高档应用可以考虑打开，但是大量额外的计算量与获得的编码性能并不成比例。所以在后面的介绍中及在实际的应用中也确实如此，以 MPEG-4 的 SP 档级为主要介绍对象。

MPEG-4 视频编码是典型的混合编码技术，即使用变换、量化和编码三步骤对图像数据或图像差值作处理。I 帧（关键帧 Key Frame）是帧内编码，即对图像数据做编码，利用了图像的空间冗余；P 帧编码主要是利用前向预测，对宏块差值编码，利用了图像的时间冗余。统计来看，P 帧编码是主流，因为 I 帧有一定的间隔。P 帧编码中的技术核心是搜索与当前宏块的最佳匹配块，即运动估计，搜索的窗口越大，宏块越匹配，但随之运算量也就越大。运动估计技术（Motion Estimation，ME）是任何视频编码算法的核心。这里以图像大小为 352 像素×288 像素、格式为 I420、编码 I、P 帧为例介绍 MPEG-4 视频编码流程，如图 14-1 所示。

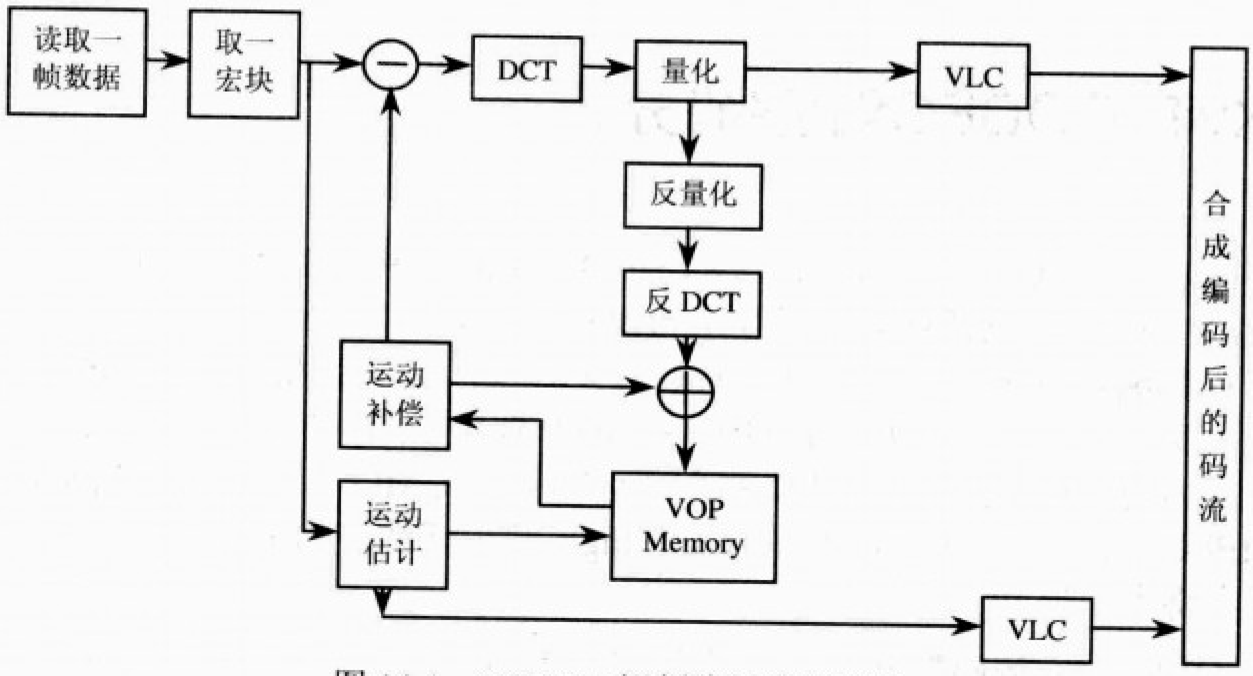


图 14-1　MPEG-4 视频编码流程框图

每个模块的功能简要介绍如下。

- 从文件中读取一帧图像，大小为 352 像素×288 像素×3/2（包含了亮度和色度）。
- 以宏块为单位进行编码，一帧图像中所有宏块编码完成，也就完成了对当前帧的编码。
- 对当前宏块进行运动估计（如果是 I 帧，则不用），根据 SAD 值决定匹配块。
- 对 8×8 块进行编码，按照其在宏块中排列的顺序进行。首先将当前宏块同参考宏块作差值，然后再对差值进行编码（I 帧不用）。
- 对当前编码块（或者经过运动补偿的差值）进行 DCT 变换。
- 对 DCT 变换后的系数进行量化。
- 反量化模块是量化模块的逆过程。
- 反 DCT 模块是 DCT 模块的逆过程。
- AC/DC 预测，即在编码 I 帧时，对当前宏块的第一行或者第一列系数同它周围的某一块作一个差值，进一步增加零系数，降低比特率。
- 反 DCT 后的宏块作运动补偿得到当前宏块的重建值，称为重建宏块。
- 码流合并模块，形成最后的视频编码码流。码流由码流头开始，然后是具体的帧。在编码之前，模块首先向输出流文件中写入码流头信息。然后写入第一帧的内容，每一帧同样由帧头信息开始，帧头信息同当前的编码内容是紧密相关的，后面的帧数据必须完全符合它规定的一些特性，譬如当前帧的量化值、运动补偿的搜索范围等。接下来是具体的帧数据，并且是按照宏块组织的。宏块内容包含当前宏块的编码信息，例如当前宏块是否编码、编码类型等，接着是运动矢量的数据，最后是具体的 6 个块的数据。

XviD CODEC 为了检测运行平台的 CPU，在编/解码器初始化前，给开发者提供了可以指定平台的接口，另外程序也能够自动检测 CPU。然后针对不同的平台初始化核心模块的函数指针，编码和解码前都要做该项工作。

```
//..................
xvid_gbl_init.cpu_flags = 0; //程序自动检测
```

```
xvid_global(NULL, XVID_GBL_INIT, &xvid_gbl_init, NULL);
    //...............
int xvid_global(void *handle, int opt, void *param1, void *param2)
{
switch(opt) {
    case XVID_GBL_INIT :             //初始化内部的函数指针、VLC码表和图像空间转换函数
        return xvid_gbl_init((xvid_gbl_init_t*)param1);
      case XVID_GBL_INFO :           //返回编码器的一些信息通知给主机
         return xvid_gbl_info((xvid_gbl_info_t*)param1);
    case XVID_GBL_CONVERT :          //颜色转换
        return xvid_gbl_convert((xvid_gbl_convert_t*)param1);
    default :
        return XVID_ERR_FAIL;
}
}
```

上述代码在调用初始化编码前调用一次，主要是初始化内部的函数指针，如通过检测 CPU 平台，使用 MMX 或 SSE 指令优化的函数初始化函数指针，创建 VLC 熵编码的码表等。主要函数为 xvid_gbl_init。

XviD 的 MPEG-4 编码算法实现以一个函数及其 3 个不同的参数传递来完成。

```
/*---------------------------编码器操作--------------------------------*/
#define XVID_ENC_CREATE  0    /* 创建编码器实例;0代表成功*/
#define XVID_ENC_DESTROY 1    /* 销毁编码器实例;0代表成功*/
#define XVID_ENC_ENCODE  2    /* 编码一帧图像: 返回编码后的字节数
                        * 0表示该帧不用写入文件（如编码器滞后或延迟） */
/*---------------------------编码器入口函数------------------------------*/
extern int xvid_encore(void *handle, int opt, void *param1, void *param2);
```

编码器的入口函数 xvid_encore 的实现如下：

```
int xvid_encore(void *handle, int opt, void *param1, void *param2)
{
switch (opt) {
case XVID_ENC_ENCODE:    //XviD编码一帧图像
    return enc_encode((Encoder *) handle,(xvid_enc_frame_t *) param1,(xvid_enc_
stats_t *) param2);
case XVID_ENC_CREATE:    //XviD创建编码器
    return enc_create((xvid_enc_create_t *) param1);
case XVID_ENC_DESTROY:   //XviD销毁编码器
    return enc_destroy((Encoder *) handle);
default:
    return XVID_ERR_FAIL;
}
}
```

上述程序是编码器的所有功能函数，创建编码器实例、使用编码器编码图像、销毁编码器。在应用层中，编码器可以有多个，这就通过编码句柄 handle 来控制不同的编码器。创建

编码器 enc_create、销毁编码器 enc_destroy 均只调用一次，循环调用 enc_encode 编码图像帧。

下面就分别详细介绍这 3 个函数的功能和实现过程。

1）XviD 创建编码器

创建 MPEG-4 编码器，首先创建编码器实例句柄，然后在该句柄下执行对编码器的参数配置、图像参数获取和空间申请等工作，这样在多路编码时，可以通过句柄来控制每一路的编码。

函数 enc_create()流程如图 14-2 所示。

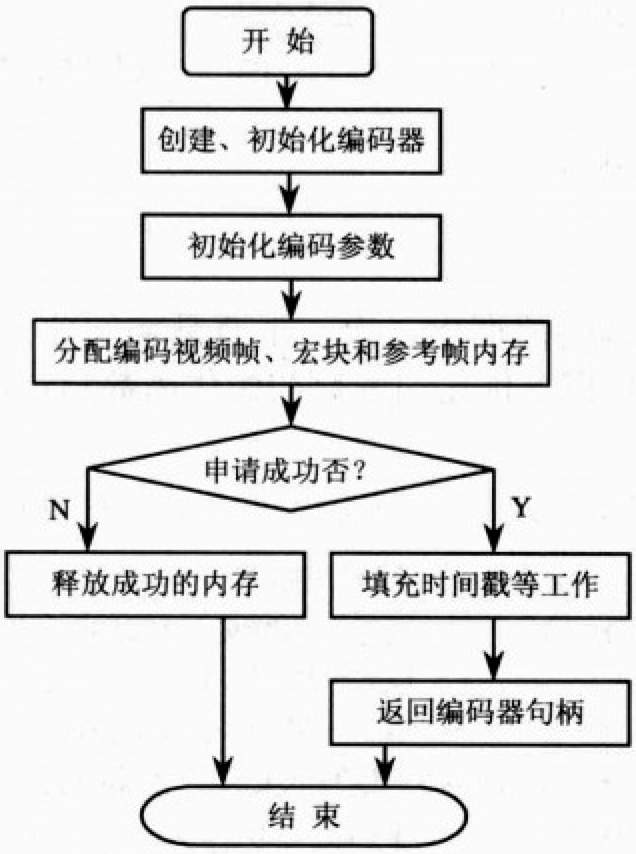


图 14-2　enc_create()流程图

根据该流程，具体代码如下。

```
int  enc_create(xvid_enc_create_t * create)
{
Encoder *pEnc;
/* 创建编码器指针，注意内存对齐*/
pEnc = (Encoder *) xvid_malloc(sizeof(Encoder), CACHE_LINE);
if (pEnc == NULL)    return XVID_ERR_MEMORY;
memset(pEnc, 0, sizeof(Encoder));                    /*初始化编码器句柄，所有指针清空*/
pEnc->mbParam.profile = create->profile;
pEnc->mbParam.global_flags = create->global;         /*编码器全局参数标志 */
pEnc->mbParam.width = create->width;                 /*视频图像的宽、高 */
pEnc->mbParam.height = create->height;
pEnc->mbParam.mb_width = (pEnc->mbParam.width + 15) / 16;   /*图像的宏块宽度*/
pEnc->mbParam.mb_height = (pEnc->mbParam.height + 15) / 16;/*图像的宏块高度*/
/*扩展后的视频图像大小 */
pEnc->mbParam.edged_width = 16 * pEnc->mbParam.mb_width + 2 * EDGE_SIZE;
pEnc->mbParam.edged_height = 16 * pEnc->mbParam.mb_height + 2 * EDGE_SIZE;
```

```
    pEnc->mbParam.fincr = MAX(create->fincr, 0);                          /*帧率递增模式*/
    pEnc->mbParam.fbase = create->fincr <= 0 ? 25 : create->fbase;     /*帧率大小*/
    pEnc->mbParam.frame_drop_ratio = MAX(create->frame_drop_ratio, 0);/*丢帧率*/
    /*最大关键帧间隔，即I帧间的最大间隔，如果没有设置则默认250*/
    pEnc->mbParam.iMaxKeyInterval = create->max_key_interval <= 0 ? (10 *
(int)pEnc->mbParam.fbase) / (int)pEnc->mbParam.fincr : create->max_key_interval;
    /*分配当前编码帧和参考帧空间 */
    pEnc->current = xvid_malloc(sizeof(FRAMEINFO), CACHE_LINE);
    pEnc->reference = xvid_malloc(sizeof(FRAMEINFO), CACHE_LINE);
    if (pEnc->current == NULL || pEnc->reference == NULL)
        goto xvid_err_memory1;
    /* 为当前帧和参考帧的所有宏块结构分配空间 */
    pEnc->current->mbs =
        xvid_malloc(sizeof(MACROBLOCK) * pEnc->mbParam.mb_width *
                    pEnc->mbParam.mb_height, CACHE_LINE);
    pEnc->reference->mbs =
        xvid_malloc(sizeof(MACROBLOCK) * pEnc->mbParam.mb_width *
                   pEnc->mbParam.mb_height, CACHE_LINE);
    if (pEnc->current->mbs == NULL || pEnc->reference->mbs == NULL)
        goto xvid_err_memory2;
    /*图像指针清空*/
    image_null(&pEnc->current->image);
    image_null(&pEnc->reference->image);
    image_null(&pEnc->vInterH);
    image_null(&pEnc->vInterV);
    image_null(&pEnc->vInterHV);
    /* 创建当前视频帧图像空间 */
    if (image_create(&pEnc->current->image, pEnc->mbParam.edged_width,pEnc->mbParam.
edged_height) < 0)
    goto xvid_err_memory3;
    /* 创建参考视频帧图像空间 */
    if(image_create(&pEnc->reference->image,pEnc->mbParam.edged_width,pEnc->
mbParam.edged_height)< 0)
    goto xvid_err_memory3;
    /* 创建vInterH参考帧图像空间 */
    if  (image_create(&pEnc->vInterH,  pEnc->mbParam.edged_width,  pEnc->mbParam.
edged_height) < 0)
        goto xvid_err_memory3;
    /* 创建vInterV参考帧图像空间 */
    if  (image_create(&pEnc->vInterV,  pEnc->mbParam.edged_width,  pEnc->mbParam.
edged_height) < 0)
        goto xvid_err_memory3;
    /* 创建vInterHV参考帧图像空间 */
    if  (image_create(&pEnc->vInterHV,  pEnc->mbParam.edged_width,  pEnc->mbParam.
edged_height) < 0)
        goto xvid_err_memory3;
```

```
    /* 初始化B帧编码序列参数*/
    pEnc->queue_head = 0;
    pEnc->queue_tail = 0;
    pEnc->queue_size = 0;
    pEnc->queue = xvid_malloc((1) * sizeof(QUEUEINFO),CACHE_LINE);
    /* 图像编码序列  */
    if(image_create(&pEnc->queue[0].image,pEnc->mbParam.edged_width,pEnc->
mbParam.edged_height) < 0)
    goto xvid_err_memory5;
    /*时间戳清零*/
    pEnc->mbParam.m_stamp = 0;
    pEnc->current->stamp = 0;
    pEnc->reference->stamp = 0;
    pEnc->iFrameNum = 0;                  /*编码图像帧数计数器清零*/
    create->handle = (void *) pEnc;   /*返回编码器的句柄 */
    return 0;                             /* ok */
    /*如果出错的话，释放以前分配的内存*/
    xvid_err_memory5:
    image_destroy(&pEnc->queue[0].image,pEnc->mbParam.edged_width,pEnc->mbParam.
edged_height);
    xvid_free(pEnc->queue);
    /* 释放图像空间*/
    image_destroy(&pEnc->current->image, pEnc->mbParam.edged_width,pEnc->mbParam.
edged_height);
    image_destroy(&pEnc->reference->image, pEnc->mbParam.edged_width,pEnc->mbParam.
edged_height);
    image_destroy(&pEnc->vInterH,  pEnc->mbParam.edged_width,pEnc->mbParam.edged_
height);
    image_destroy(&pEnc->vInterV,  pEnc->mbParam.edged_width,pEnc->mbParam.edged_
height);
    image_destroy(&pEnc->vInterHV, pEnc->mbParam.edged_width,pEnc->mbParam.edged_
height);
    xvid_err_memory2:
    /* 释放当前帧和参考帧的mbs宏块内存*/
    xvid_free(pEnc->current->mbs);
    xvid_free(pEnc->reference->mbs);
    xvid_err_memory1:
    /* 释放当前帧和参考帧的内存*/
    xvid_free(pEnc->current);
    xvid_free(pEnc->reference);
    /* 释放编码器*/
    xvid_free(pEnc);
    xvid_err_memory3:
    create->handle = NULL;
    return XVID_ERR_MEMORY;
    }
```

上述代码完成编码器的创建、编码参数的初始化、工作空间的申请等。编码器的初始化函数可以不用过多地考虑优化，因为这个函数只调用一次，不像编码函数多次循环地调用。

2）XviD 编码一帧图像

对输入的一帧编码图像，根据程序判定或用户强制类型进行编码。编码器返回编码后的码流和长度及编码信息的统计。

函数 enc_encode()流程如图 14-3 所示。

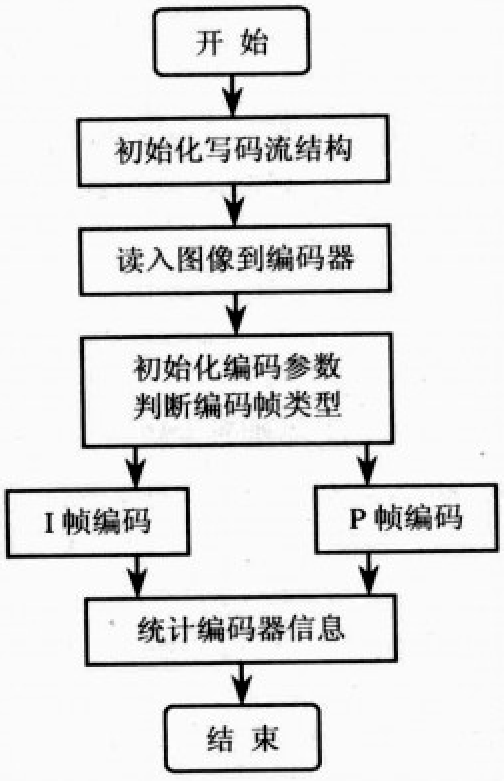


图 14-3 enc_encode()流程图

根据该流程，具体代码如下。

```
int enc_encode(Encoder * pEnc, xvid_enc_frame_t * xFrame, xvid_enc_stats_t * stats)
{
xvid_enc_frame_t * frame;
int type;
Bitstream bs;                //存储码流
xFrame->out_flags = 0;
//初始化码流指针，以便于存储编码后的码流
BitstreamInit(&bs, xFrame->bitstream, 0);
//图像颜色空间为I420，即YUV的比例是4:2:0（或4:1:1）
if (xFrame->input.csp != XVID_CSP_NULL)
{
    q = &pEnc->queue[0];
    //读入图像帧到编码器的序列组中，input.plane => q->image
    if (image_input(&q->image, pEnc->mbParam.width, pEnc->mbParam.height,
        pEnc->mbParam.edged_width, (uint8_t**)xFrame->input.plane,
        xFrame->input.stride, xFrame->input.csp, 0))
    {
        return XVID_ERR_FORMAT;            //图像格式错误
    }
```

```
        q->frame = *xFrame;
        pEnc->queue_tail = 0;
        pEnc->queue_size++;
    }
    /*————————————分析图像编码队列———————— */
    if (pEnc->queue_size == 0){                        /*队列为空*/
        if (xFrame->input.csp == XVID_CSP_NULL)   /*没有图像继续输入*/
            return XVID_ERR_END;                        /*编码完毕，没有图像输出*/
        goto done;                                       /*没有图像编码;编码器延迟*/
    }
    SWAP(FRAMEINFO*, pEnc->current, pEnc->reference);          /*交换两图像的指针*/
    image_swap(&pEnc->current->image, &pEnc->queue[0].image);  /*从序列组中读入一帧到
                                                                    编码器中*/
    frame = &pEnc->queue[0].frame;
    pEnc->queue_head = 0;
    pEnc->queue_size--;
    /*————————————初始化编码器的当前帧编码参数————————————*/
    pEnc->current->fincr=pEnc->mbParam.fincr>0?pEnc->mbParam.fincr:frame->fincr;
    inc_frame_num(pEnc);                                   /*修改时间戳、编码计数器*/
    pEnc->current->vol_flags = frame->vol_flags;          /*获取输入的VOL标志*/
    pEnc->current->vop_flags = frame->vop_flags;          /*获取输入的VOP标志*/
    pEnc->current->motion_flags = frame->motion;          /*获取输入的Motion标志*/
    pEnc->current->fcode = pEnc->mbParam.m_fcode;         /*获取ME的搜索窗口*/
    /*————————————选择编码帧类型和量化————————————*/
    type = frame->type;
    pEnc->current->quant = frame->quant;
    if (type > 0){
        type = type2coding(type);
    } else{                 /* XVID_TYPE_AUTO */
        if (pEnc->iFrameNum == 0 || (pEnc->mbParam.iMaxKeyInterval > 0
            && pEnc->iFrameNum >= pEnc->mbParam.iMaxKeyInterval))
         {
            pEnc->iFrameNum = 0;
            type = I_VOP;          //强制编码类型I帧编码
        }else{
            type = P _VOP;         //强制编码类型P帧编码
        }
    }
    if (type != I_VOP)
        pEnc->current->vol_flags = pEnc->mbParam.vol_flags;
      /*编码器的编码帧计数*/
    pEnc->iFrameNum++;
    if (type == I_VOP) {
        pEnc->iFrameNum = 1;
        pEnc->mbParam.vol_flags = pEnc->current->vol_flags;   /*更新I_VOP的VOL标志*/
          FrameCodeI(pEnc, &bs);                               /*图像为I帧编码 */
```

```
    xFrame->out_flags |= XVID_KEYFRAME;
} else { /* (type == P_VOP || type == S_VOP) */
    FrameCodeP(pEnc, &bs);                              /*图像为P帧编码 */
}
    /*为统计编码器的各个参数，保存当前帧的编码参数：编码类型、量化步长、*/
    /*码流长度、被编码宏块和没有编码的宏块数目等                          */
#if 1
stats->type      = pEnc->current->coding_type+1;  //帧编码类型:I/P/B
stats->quant= pEnc->current->quant;               //量化步长
stats->vol_flags= pEnc->current->vol_flags;       //VOL标志
stats->vop_flags= pEnc->current->vop_flags;       //VOP标志
stats->length    = pEnc->current->length;         //当前帧编码后的码流长度
stats->kblks= pEnc->current->sStat.kblks;         //以intra模式编码的宏块数目
stats->mblks= pEnc->current->sStat.mblks;         //以inter模式编码的宏块数目
stats->ublks= pEnc->current->sStat.ublks;         //未编码的宏块数目
#endif
done:
return BitstreamLength(&bs);      /*编码完毕，计算压缩后的码流长度*/
}
```

上述代码实现对传入的一帧图像的编码，编码方式可以是外部强制定义，也可以根据 I 帧间隔计算当前帧的编码类型。从上述代码分析可知，XviD 的 MPEG-4 SP 编码算法有 2 个函数 FrameCodeI 和 FrameCodeP 来实现真正的图像编码。

① FrameCodeI

I 帧编码又称为关键帧编码，是消除图像的空间冗余。I 帧编码与静态图像的编码算法与 JPEG 有些类似。

I 帧编码流程如图 14-4 所示。

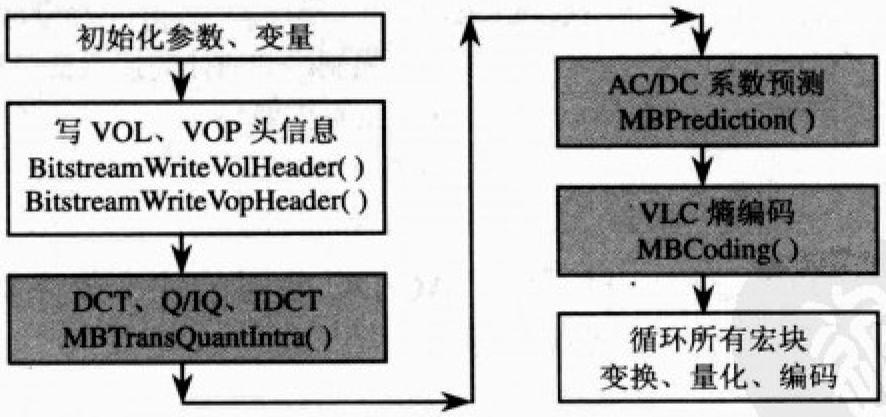


图 14-4　I 帧编码流程框图

根据该流程，I 帧编码代码实现如下。

```
int FrameCodeI(Encoder * pEnc, Bitstream * bs)
{
int bits = BitstreamPos(bs);
uint32_t mb_width     = pEnc->mbParam.mb_width;            /*宏块宽*/
uint32_t mb_height    = pEnc->mbParam.mb_height;           /*宏块高*/
uint32_t edged_width     = pEnc->mbParam.edged_width;      /*扩展后的图像宽度*/
```

```
    DECLARE_ALIGNED_MATRIX(dct_coeff , 6, 64, int16_t, CACHE_LINE);/*DCT系数*/
    DECLARE_ALIGNED_MATRIX(qnt_coeff , 6, 64, int16_t, CACHE_LINE);/*DCT系数量化结
                                                                    果*/
    uint16_t x, y;
    pEnc->current->coding_type = I_VOP;                  /*当前边编码类型I_VOP*/
    pEnc->current->mbs[0].quant = pEnc->current->quant;  /*获取量化步长*/
    SetMacroblockQuants(&pEnc->mbParam, pEnc->current);  /*设置每个宏块的量化大小*/
    BitstreamWriteVolHeader(bs, &pEnc->mbParam, pEnc->current);/*写VOL头信息*/
    set_timecodes(pEnc->current,pEnc->reference,pEnc->mbParam.fbase);
    BitstreamPad(bs);                                    /*写VOP头信息*/
    BitstreamWriteVopHeader(bs, &pEnc->mbParam, pEnc->current, 1, pEnc->current->
mbs[0].quant);
        /*初始化统计编码信息*/
    pEnc->current->sStat.iTextBits = 0;
    pEnc->current->sStat.kblks = mb_width * mb_height;
    pEnc->current->sStat.mblks = pEnc->current->sStat.ublks = 0;
    for (y = 0; y < mb_height; y++)
        for (x = 0; x < mb_width; x++) {
            MACROBLOCK *pMB = &pEnc->current->mbs[x + y * pEnc->mbParam.mb_width];
            CodeIntraMB(pEnc, pMB);
            /*图像数据DCT变换、量化、反量化、IDCT变换，更新当前图像数据*/
              MBTransQuantIntra(&pEnc->mbParam, pEnc->current, pMB, x, y,dct_coeff,
qnt_coeff);          /*AC/DC预测*/
            MBPrediction(pEnc->current, x, y, pEnc->mbParam.mb_width, qnt_coeff);
            /*系数VLC编码*/
            MBCoding(pEnc->current, pMB, qnt_coeff, bs, &pEnc->current->sStat);
        }
    BitstreamPadAlways(bs);                  /*填充对齐*/
    pEnc->current->length = (BitstreamPos(bs) - bits) / 8; /*获取当前帧编码后的码流长速*/
    pEnc->mbParam.m_fcode = 1;               /*强制运动估计的搜索窗口*/
    pEnc->current->is_edged = 0;             /*没有扩展*/
    return 1;                                /* intra */
    }
```

I 帧编码比较简单，编码过程一目了然。AC/DC 是对宏块变换系数的第一行和第一列作预测，以进一步增加零系数的数目，提高压缩比。

② FrameCodeP

P 帧编码是帧间编码，利用前面已经编码、解码重建的帧作为参考，插值出多个参考帧，在这些参考帧中搜索最佳匹配块。将当前编码块和匹配块相减，对残差做 DCT、量化和编码。不同的搜索算法、不同的搜索窗口，搜索的匹配块的运动向量（MV）、匹配相似度等有不同。

在图像帧 P 帧编码中，有一定数量的 Intra 模式的块编码，其他绝大部分为 Inter 模式或 not_coded 不编码模式。算法支持当整整的 Intra 块编码模式超过一定数量时，强制为 I 帧编码。

Inter 为帧间编码，即对残差进行编码。not_coded 表示当前宏块没有编码，为零块，则

在解码重建时，直接把参考帧的块复制过来作为解码图像，这种块模式的宏块一般都是背景图像。

P 帧编码 FrameCodeP()的流程如图 14-5 所示。

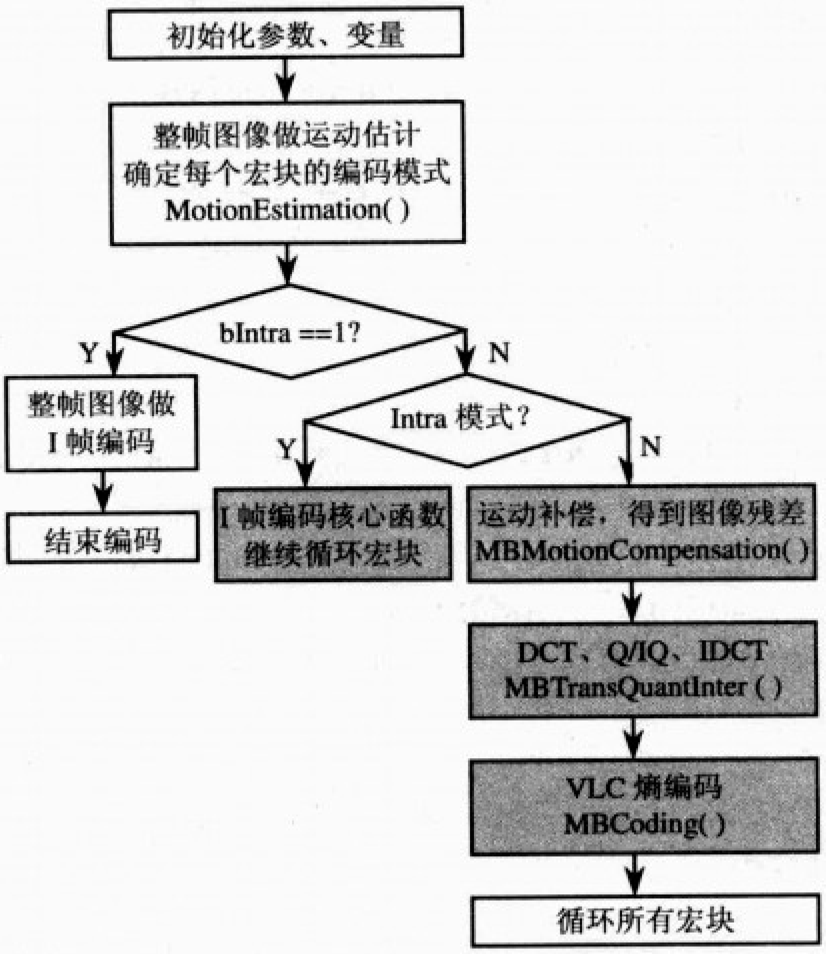


图 14-5　P 帧编码流程框图

根据该流程，P 帧编码代码实现如下。

```
int FrameCodeP(Encoder * pEnc,Bitstream * bs)
{
int bits = BitstreamPos(bs);
int iLimit,x,y;
int bIntra=0, skip_possible;
FRAMEINFO *const current = pEnc->current;                    /*当前编码帧*/
FRAMEINFO *const reference = pEnc->reference;                /*重建参考帧*/
MBParam * const pParam = &pEnc->mbParam;                     /*宏块编码结构信息*/
const uint32_t mb_width = pParam->mb_width;                  /*宏块宽度*/
const uint32_t mb_height = pParam->mb_height;                /*宏块高度*/
const uint32_t edged_width = pParam->edged_width;            /*扩展图像宽度*/
DECLARE_ALIGNED_MATRIX(dct_coeff , 6, 64, int16_t, CACHE_LINE); /*DCT系数*/
DECLARE_ALIGNED_MATRIX(qnt_coeff , 6, 64, int16_t, CACHE_LINE); /*DCT系数量化结果*/
pParam->m_rounding_type = 1 - pParam->m_rounding_type;
current->rounding_type = pParam->m_rounding_type;
current->fcode = pParam->m_fcode;
current->coding_type = P_VOP;
pEnc->current->mbs[0].quant = pEnc->current->quant;
SetMacroblockQuants(&pEnc->mbParam, current);          /*初始化宏块量化步长*/
/*运动估计模块，把当前图像与上一帧的重建帧做最佳匹配，根据SAD抉择*/
```

```
    iLimit = (mb_width * mb_height)>>1;          /*当Intra块超过宏块的50%时，强制I帧编码*/
    bIntra = MotionEstimation(&pEnc->mbParam, current, reference,iLimit);
    if (bIntra == 1) return FrameCodeI(pEnc, bs);/* bIntra为1则执行I帧编码，如场景切换*/
    set_timecodes(current,reference,pParam->fbase);
        /*写VOP头信息到码流*/
    BitstreamWriteVopHeader(bs, &pEnc->mbParam, current, 1, current->mbs[0].quant);
        /*初始化状态统计变量*/
    current->sStat.iTextBits = current->sStat.iMvSum = current->sStat.iMvCount =
        current->sStat.kblks = current->sStat.mblks = current->sStat.ublks = 0;
        /*以宏块为单位循环编码*/
        for (y = 0; y < mb_height; y++) {
        for (x = 0; x < mb_width; x++) {
            MACROBLOCK *pMB = &current->mbs[x + y * mb_width];
            bIntra = (pMB->mode == MODE_INTRA);
            if (bIntra){                          /*Intra块编码，即I帧编码的核心函数*/
                CodeIntraMB(pEnc, pMB);
                MBTransQuantIntra(&pEnc->mbParam, current, pMB, x, y, dct_coeff,
qnt_coeff);
                MBPrediction(current, x, y, mb_width, qnt_coeff);
                MBCoding(current, pMB, qnt_coeff, bs, &current->sStat);
                continue;
            }
            /*运动补偿，根据运动向量确定具体的宏块参考位置，并做差值计算，结果存放dct_coeff
中*/
            MBMotionCompensation(pMB,  x,  y,  pRef,  pCur,  dct_coeff,  pParam->
edged_width);
            /*差值做DCT、量化、反量化、IDCT，并更新到当前图像*/
              if (pMB->mode != MODE_NOT_CODED)          /*差值编码，即差值不全为零*/
              pMB->cbp =MBTransQuantInter(&pEnc->mbParam, current,
                                    pMB, x, y, dct_coeff, qnt_coeff);
            skip_possible = (pMB->cbp == 0) && (pMB->mode == MODE_INTER);
            if (current->coding_type == P_VOP)
                skip_possible &= ( (pMB->mvs[0].x == 0) && (pMB->mvs[0].y == 0) );
              /*宏块没有编码，跳过该宏块，继续编码*/
            if ( (pMB->mode == MODE_NOT_CODED) || (skip_possible)) {
                pMB->mode = MODE_NOT_CODED;
                MBSkip(bs);
                continue;
            }
              /*宏块编码，即差值的DCT系数量化值做VLC编码*/
            MBCoding(current, pMB, qnt_coeff, bs, &pEnc->current->sStat);
        }
    }
    pParam->m_fcode = 1;                                    /*运动估计ME搜索窗口*/
    pEnc->current->is_edged = 0;                            /*没有做扩展*/
```

```
BitstreamPadAlways(bs);                              /*填充编码后的比特流*/
current->length = (BitstreamPos(bs) - bits) / 8;     /*编码后的码流长度*/
return 0;
```

在 P 帧编码前，首先对整帧做运动估计，确定每个宏块的编码模式：Intra、Inter、Not_coded。如果 Intra 块的数量超过宏块总数的一定比例，则强制该帧做 I 帧编码。在 P 帧编码中，既有 Inter 块的编码也有 Intra 块的编码，同时对零块不做编码 not_coded，即跳过该宏块 MBSkip。另外，为了提高编码效率，编码器中省略了图像扩展的模块。但是需要特别注意，为了保证解码器解码出的图像正确，在创建编、解码器的图像帧空间时（image_create 函数），应都初始化为确定的相同的值，否则解码器重建的图像边缘会出错。

③ Intra 宏块变换、量化/反量化、反变换

图像 Intra 块的变换与量化一般是放置在一起的。变换采用离散余弦变换（DCT），量化采用 H.263 的均匀量化。并且量化采取的是查表、移位的方式，从而避免了除法。实现该功能的函数是 MBTransQuantIntra()，其代码如下。

```
void MBTransQuantIntra(const MBParam * const pParam, const FRAMEINFO * const frame,
                       MACROBLOCK * const pMB, const uint32_t x_pos, const uint32_t y_pos,
                       int16_t data[6 * 64], int16_t qcoeff[6 * 64])
{
/* 从图像空间读取宏块数据，并做16位扩展 */
MBTrans8to16(pParam, frame, pMB, x_pos, y_pos, data);
/* 宏块做DCT变换 */
MBfDCT(pParam, frame, pMB, x_pos, y_pos, data);
/* 宏块DCT系数做量化 */
MBQuantIntra(pParam, frame, pMB, data, qcoeff);
/* 宏块DCT系数做反量化 */
MBDeQuantIntra(pParam, pMB->quant, data, qcoeff);
/* 宏块做IDCT变换 */
MBiDCT(data, 0x3F);
    /* 把重建的宏块更新到当前编码帧，直接复制*/
MBTrans16to8(pParam, frame, pMB, x_pos, y_pos, data, 0, 0x3F);
}
```

上述程序实现宏块（包括一个 16×16 的亮度块，两个 8×8 的色度块）的变换与量化处理。首先扩展图像为 16 位，为 DCT 做数据源，接着做变换和量化。最后为保证与解码图像一致，做反 DCT 和反量化，将这些重建值更新到编码图像。

④ Inter 宏块变换、量化/反量化、反变换

Inter 块的处理与 Intra 块基本相同，只是根据 CBP 编码模式决定哪些块的重建值更新编码图像。实现函数为 MBTransQuantInter()，其代码实现如下。

```
uint8_t MBTransQuantInter(const MBParam * const pParam, const FRAMEINFO * const frame,
                       MACROBLOCK * const pMB,    const uint32_t x_pos,const uint32_t
y_pos,
                       int16_t data[6 * 64], int16_t qcoeff[6 * 64])
```

```
{
uint8_t cbp;
/* MC中已经做了减法，得到了图像差值 */
/* 宏块DCT */
MBfDCT(pParam, frame, pMB, x_pos, y_pos, data);
/* Set the limit threshold */
/* 宏块DCT系数量化 */
cbp = MBQuantInter(pParam, frame, pMB, data, qcoeff, 0, 1);
/* 宏块DCT系数反量化 */
MBDeQuantInter(pParam, pMB->quant, data, qcoeff, cbp);
/* 宏块反DCT变换 */
MBiDCT(data, cbp);
    /* 重建的宏块数据更新到当前编码帧，做加操作，与MC相反 */
MBTrans16to8(pParam, frame, pMB, x_pos, y_pos, data, 1, cbp);
return(cbp); /* 返回块编码模式CBP (Coded Block Pattern) */
}
```

上述代码实现 Inter 宏块的变换与量化处理，需特别注意 CBP 的获取与应用。

⑤ 变换、量化后的系数做 VLC 熵编码。

```
void MBCoding(const FRAMEINFO * const frame,  MACROBLOCK * pMB,
        int16_t qcoeff[6 * 64], Bitstream * bs, Statistics * pStat)
{
if (frame->coding_type != I_VOP)
    BitstreamPutBit(bs, 0);                          /* 编码模式: not_coded */
if (pMB->mode == MODE_INTRA)
    CodeBlockIntra(frame, pMB, qcoeff, bs, pStat); /* Inter宏块编码，MV、系数VLC*/
else
    CodeBlockInter(frame, pMB, qcoeff, bs, pStat); /* Intra宏块系数VLC*/
}
```

对于 Inter 宏块编码，首先写 mcbpc、cbpy、MV，最后写宏块系数。对于 Intra 宏块编码，首先写 mcbpc、AC 预测方向、cbpy，最后写宏块系数。对于图像或图像差值做变换、量化/反量化、反变换、编码等底层核心模块，为了提高效率，一般要汇编优化、改写。在后面的两节中，分别介绍采用 MMX/SSE 2 和 DM642 两个平台优化核心模块。这也符合视频编码算法开发应用的思路：框架用 C 语言编程、核心模块用汇编优化。

3）XviD 销毁编码器

销毁编码器实例对应于创建编码器实例的工作，即把所有申请的内存释放。

```
int enc_destroy(Encoder * pEnc)
{
/*释放图像序列空间*/
image_destroy(&pEnc->queue[0].image, pEnc->mbParam.edged_width,pEnc->mbParam.
edged_height);
xvid_free(pEnc->queue);
    /*释放当前编码帧的图像空间*/
```

```
    image_destroy(&pEnc->current->image, pEnc->mbParam.edged_width,pEnc->mbParam.
edged_height);
        /*释放参考帧的图像空间*/
    image_destroy(&pEnc->reference->image, pEnc->mbParam.edged_width,pEnc->mbParam.
edged_height);
        /*释放参考帧的水平插值帧的图像空间*/
    image_destroy(&pEnc->vInterH,  pEnc->mbParam.edged_width,pEnc->mbParam.edged_
height);
        /*释放参考帧的垂直插值帧的图像空间*/
    image_destroy(&pEnc->vInterV,  pEnc->mbParam.edged_width,pEnc->mbParam.edged_
height);
        /*释放参考帧的水平、垂直（即斜角方向）插值帧的图像空间*/image_destroy
(&pEnc->vInterHV, pEnc->mbParam.edged_width,pEnc->mbParam.edged_height);
    /*释放宏块编码空间*/
    xvid_free(pEnc->current->mbs);
    xvid_free(pEnc->current);     /*释放空间*/
    /*释放宏块编码空间*/
    xvid_free(pEnc->reference->mbs);
    xvid_free(pEnc->reference);  /*释放空间*/
    xvid_free(pEnc);             /*释放编码器的空间*/
    return 0;
    }
```

XviD CODEC 的 MPEG-4 视频编码有多种参数可供选择。开发者可以根据自己的指标要求和不同的实现平台，组合编码参数或删减无用的代码以提高编码器的效率。一个典型的嵌入式 MPEG-4 视频编码算法一般不支持 B 帧编码、1/4 像素精度或 MPEG 量化模式，但是可采用浮点的码流控制机制以适应网络传输。对于通用 CPU 如 Intel 系列 CPU，在实现视频编码以验证算法的效率时，可以选择较多的功能，但是实时性可能不高。不过，随着 CPU 主频等性能的提升，软压缩得到了极大的发展，对硬件视频压缩有一定的影响。

2. MPEG-4 视频解码

视频解码工作相对于视频编码，运算量大大降低，在一般情况下，后者是前者的 1/3。大家都知道，MPEG-4 视频算法是可兼容传统视频帧的面向对象编码的。但是目前视频对象分割是难点，因此几乎所有的 MPEG-4 视频应用都是基于传统的矩形视频帧来处理的。只不过 MPEG-4 把整个图像画面看做一个对象，即视频对象平面 VOP（Video Object Plane）。

如图 14-6 所示是 VOP 解码功能框图，频码流包含 I-VOP、B-VOP、P-VOP 纹理信息和运动信息。帧间（Inter）和帧内（Intra）解码处理单位仍然是 16×16 的宏块。其中 Intra 宏块是图像数据经过 IDCT 和 AC/DC 预测得到的。Inter 块是块数据间的残差作 IDCT，然后加回到比较块，比较块的位置是根根码流中的运动向量 MV（1 个或 4 个）的值来确定的。运动向量的精度有 1/2 像素、1/4 像素，这根据码流中的信息来决定，当然有可能指向参考帧的边界外。

XviD CODEC 解码算法也提供了对 CPU 平台的汇编优化支持。检测 CPU、初始化核心模块函数指针的代码完全编码工作与相同，请参考前面的小节。

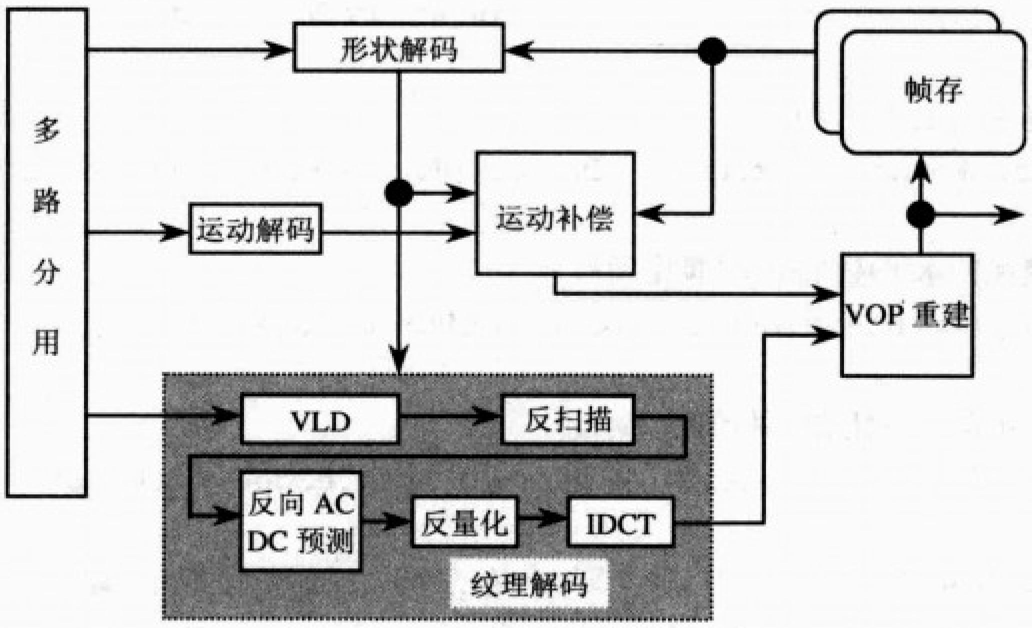


图 14-6　VOP 解码功能框图

XviD 的 MPEG-4 解码算法实现以一个函数及其 3 个不同的参数传递来完成。

```
/*--------------------------解码器操作--------------------------------------*/
#define XVID_DEC_CREATE  0    /* 创建解码器实例; 0代表成功*/
#define XVID_DEC_DESTROY 1    /* 销毁解码器实例; 0代表成功*/
#define XVID_DEC_ENCODE  2    /* 解码一帧图像: 返回已经解码的字节数 (>=0) */
/*----------------------------编码器入口函数----------------------------------*/
extern int xvid_decore(void *handle, int opt, void *param1, void *param2);
```

解码器的入口函数 xvid_decore 的实现如下：

```
int xvid_decore(void *handle, int opt, void *param1, void *param2)
{
switch (opt) {
case XVID_DEC_CREATE:                /*创建解码器实例*/
    return decoder_create((xvid_dec_create_t *) param1);
case XVID_DEC_DESTROY:               /*销毁解码器实例*/
    return decoder_destroy((DECODER *) handle);
case XVID_DEC_DECODE:                /*解码一帧图像*/
    return decoder_decode((DECODER *) handle,          /*解码器句柄*/
                  (xvid_dec_frame_t *) param1,    /*解码帧结构体*/
                  (xvid_dec_stats_t*) param2);    /*解码状态结构体*/
default:
    return XVID_ERR_FAIL;
}
}
```

上述代码是解码器的所有功能函数，创建解码器实例、使用解码器解码图像、销毁解码器。在应用层，解码器可以有多个，这就通过解码句柄 handle 来控制不同的解码器。创建解码器 decoder_create、销毁解码器 decoder_destroy 均只调用一次。循环调用 decoder_decode 解码图像帧。

1）创建解码器实例

创建解码器实例，即解码器句柄。解码器的所有动作和使用配置都是通过该句柄来控制完成的，XviD CODEC 视频解码器可以同时解码多路视频。

```
/*图像分辨率大小发生变化，释放已申请的内存，并重新申请*/
static int decoder_resize(DECODER * dec)
{
/*释放存在的图像空间*/
image_destroy(&dec->cur, dec->edged_width, dec->edged_height);
image_destroy(&dec->refn[0], dec->edged_width, dec->edged_height);
image_destroy(&dec->refn[1], dec->edged_width, dec->edged_height);
image_destroy(&dec->tmp, dec->edged_width, dec->edged_height);
/*所有图像指针清零*/
image_null(&dec->cur);          //当前解码帧的空间y/u/v指针清空
 image_null(&dec->refn[0]);     //参考帧0的空间y/u/v指针清空
 image_null(&dec->refn[1]);     //参考帧1的空间y/u/v指针清空
 image_null(&dec->tmp);         //tmp临时空间的y/u/v指针清空
 xvid_free(dec->last_mbs);      //释放宏块结构空间
 xvid_free(dec->mbs);           //释放宏块结构空间
 dec->last_mbs = NULL;          //指针清空
 dec->mbs = NULL;               //指针清空
/*重新申请内存*/
dec->mb_width = (dec->width + 15) / 16;           /*图像帧宏块宽度，16的倍数*/
dec->mb_height = (dec->height + 15) / 16;         /*图像帧宏块高度，16的倍数*/
dec->edged_width = 16 * dec->mb_width + 2 * EDGE_SIZE;/*图像边扩展了的图像宽度*/
dec->edged_height = 16 * dec->mb_height + 2 * EDGE_SIZE;/*图像边扩展了的图像高度*/
    /*申请图像空间*/
if (  image_create(&dec->cur, dec->edged_width, dec->edged_height)
   || image_create(&dec->refn[0], dec->edged_width, dec->edged_height)
   || image_create(&dec->refn[1], dec->edged_width, dec->edged_height)
   || image_create(&dec->tmp, dec->edged_width, dec->edged_height))
    goto memory_error;          //申请失败
    /*申请所有宏块的解码信息*/
dec->mbs = xvid_malloc(sizeof(MACROBLOCK) * dec->mb_width * dec->mb_height,
CACHE_LINE);
if (dec->mbs == NULL)  goto memory_error;         //申请失败
memset(dec->mbs, 0, sizeof(MACROBLOCK) * dec->mb_width * dec->mb_height); /*
清空*/
/* 解码B帧时，用到的宏块解码信息 */
dec->last_mbs=xvid_malloc(sizeof(MACROBLOCK)*dec->mb_width * dec->mb_height,
CACHE_LINE);
if (dec->last_mbs == NULL)  goto memory_error;  //申请失败
memset(dec->last_mbs, 0, sizeof(MACROBLOCK) * dec->mb_width * dec->mb_height);
return 0;
memory_error:
       /*释放图像空间、结构体，指针置空 */
  xvid_free(dec->mbs);
  image_destroy(&dec->cur, dec->edged_width, dec->edged_height);
//释放当前解码帧空间
  image_destroy(&dec->refn[0], dec->edged_width, dec->edged_height);
//释放参考帧0空间
```

```
    image_destroy(&dec->refn[1], dec->edged_width, dec->edged_height);
  //释放参考帧1空间
    image_destroy(&dec->tmp, dec->edged_width, dec->edged_height);
  //释放临时帧空间
    /*释放解码器句柄*/
    xvid_free(dec);
    return XVID_ERR_MEMORY;
  }
  /*创建编码器实例*/
  int decoder_create(xvid_dec_create_t * create)
  {
    DECODER *dec;
   dec = xvid_malloc(sizeof(DECODER), CACHE_LINE); /*创建并初始化解码器句柄，使句柄完
                                                     全可控*/
    if (dec == NULL)   return XVID_ERR_MEMORY;
    memset(dec, 0, sizeof(DECODER));
    create->handle = dec;        /*获取解码输入的参数*/
    dec->width = create->width;
    dec->height = create->height;
    image_null(&dec->cur);       /*图像空间指针清零*/
    image_null(&dec->refn[0]);
    image_null(&dec->refn[1]);
    image_null(&dec->tmp);
    dec->mbs = NULL;
    dec->last_mbs = NULL;
    init_postproc(&dec->postproc);       /*支持图像后处理，如滤波、去块效应*/
    dec->frames = 0;                     /*支持B帧解码，用来保存参考帧的时间*/
    dec->time = dec->time_base = dec->last_time_base = 0;
    dec->low_delay = 0;
    dec->packed_mode = 0;
    dec->time_inc_resolution = 1;        /* 直到VOL头更改为其他数据 */
    /*创建解码器时，已知图像宽度和高度*/
    dec->fixed_dimensions = (dec->width > 0 && dec->height > 0);
    if (dec->fixed_dimensions)   return decoder_resize(dec);
    else  return 0;
  }
```

上述代码实现创建解码器，把图像空间的指针清空，如果已知图像的分辨率，则申请各种空间。如果不知道分辨率或分辨率设置不正确，则解码器在分析码流的头结构后，获取分辨率再重新申请空间。

在创建解码器时，也可以不输入图像分辨率，因为码流中本身有分辨率 width、height 信息，能够从码流中解析到，然后再重新申请图像空间。

2）解码一帧图像

MPEG-4 SP 视频编码算法中，编码帧类型是 I 帧和 P 帧。码流头结构中有帧类型信息、量化信息、fcode 值、intra_dc_threshold 等。XviD 提供的 MPEG-4 视频解码在解码到 VOL 时会返回应用程序，为了应用层更直接地得到解码图像，现在修改为只有解析并解码了图像帧

数据时才返回应用程序。

解码器解码图像 decoder_decode()的流程如图 14-7 所示。根据该流程，代码实现如下。

```
    int decoder_decode(DECODER * dec, xvid_dec_frame_t * frame, xvid_dec_stats_t *
stats)
    {
    uint32_t rounding;                          /*图像饱和类型，在插值时四舍五入的饱和大小*/
    uint32_t reduced_resolution;
    uint32_t quant = 2;                         /*默认的量化步长初始化*/
    uint32_t fcode_forward;                     /*运动估计中搜索窗口的大小，前向*/
    uint32_t fcode_backward;                    /*运动估计中搜索窗口的大小，后向*/
    uint32_t intra_dc_threshold;                /*Intra块的直流DC阈值*/
    WARPPOINTS gmc_warp;
    int coding_type;                            /*当前编码帧类型*/
    if (frame->length < 0)      return XVID_ERR_END;   /*检测输入码流长度的合法性*/
    BitstreamInit(&g_bs, frame->bitstream, frame->length);  /*初始化码流结构体*/
       /*解析视频码流头结构、信息*/
    repeat:
    coding_type = BitstreamReadHeaders(&g_bs, dec, &rounding, &reduced_resolution,
            &quant,   &fcode_forward,   &fcode_backward,   &intra_dc_threshold,
&gmc_warp);
    if (coding_type == -1) {                    /* 继续解析头信息，直到解析到图像数据 */
        goto repeat;
    }
    if (coding_type == -2 || coding_type == -3) {/* 当前是VOL头或需要修改图像空间大小 */
        if (coding_type == -3) decoder_resize(dec);
        if (stats)  goto repeat;                /* 继续解析头信息，直到解析到图像数据 */
    }
        /*确保解码的第一帧是I帧，如果不是I帧则继续解析后续码流*/
    if(dec->frames == 0 && coding_type != I_VOP) goto repeat;
        if (coding_type != B_VOP)      {
        switch(coding_type) {
        case I_VOP :          /*I帧解码*/
            decoder_iframe(dec,   &g_bs,   reduced_resolution,   quant,   intra_dc_
threshold);
            break;
        case P_VOP :          /*P帧解码*/
            decoder_pframe(dec, &g_bs, rounding, reduced_resolution, quant,
                        fcode_forward, intra_dc_threshold, NULL);
            break;
        case S_VOP :          /*S帧解码*/
            decoder_pframe(dec, &g_bs, rounding, reduced_resolution, quant,
                        fcode_forward, intra_dc_threshold, &gmc_warp);
            break;
        }
        dec_image_swap(&dec->refn[0], &dec->refn[1]);  /*交换两个参考帧的指针*/
        dec_image_swap(&dec->cur, &dec->refn[0]);       /*交换当前图像和参考帧的指针*/
        dec->last_reduced_resolution = reduced_resolution;
```

```
    dec->last_coding_type = coding_type;                    /*保存当前帧类型*/
    dec->frames++;                                          /*解码帧计数器*/
  }
  done :
  return (BitstreamPos(&g_bs) + 7) / 8;                     /*已解码的码流，单位B*/
  }
```

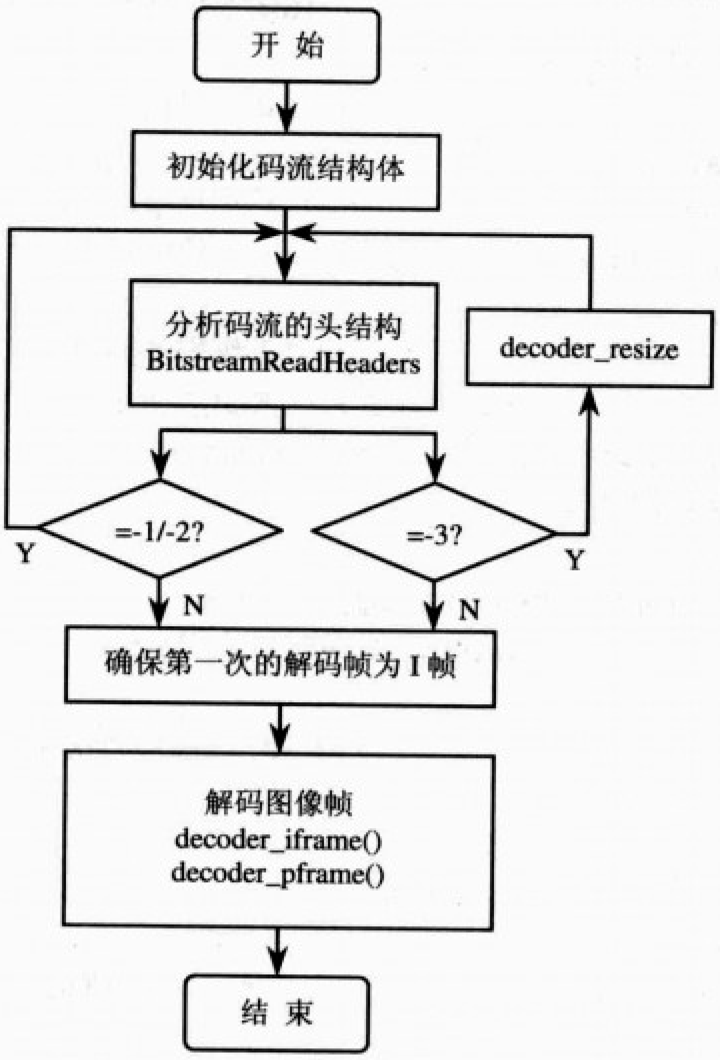


图 14-7　decoder_decode()流程图

BitstreamReadHeaders()函数根据 MPEG-4 标准码流协议解析编码信息。如果用户在编码端对 VOL、VOP 的写头信息作了修改，则在解码端的该函数中，要一一对应起来。如为了加密需要，就可以在编码端的写 VOL/VOP 信息作特定的修改。SP 档级的 MPEG-4 视频帧有 I 帧和 P 帧，本章对 I 帧和 P 帧的解码过程做剖析。I 帧解码由 decoder_iframe 函数、P 帧解码由 decoder_pframe 函数完成。

① decoder_iframe

解码是其编码的逆工作，获得 mcbpc、acpred_flag、cbpy 等，将运动向量初始化为 0。

```
  static void decoder_iframe(DECODER * dec, Bitstream * bs, int quant, int
intra_dc_threshold)
  {
   uint32_t x, y;
   const uint32_t mb_width = dec->mb_width;
   const uint32_t mb_height = dec->mb_height;
   /*循环解码所有宏块*/
   for (y = 0; y < mb_height; y++)
    for (x = 0; x < mb_width; x++) {
```

```
        MACROBLOCK *mb;
        uint32_t mcbpc, cbpc, acpred_flag, cbpy, cbp;
        mb = &dec->mbs[y * dec->mb_width + x];
        mcbpc = get_mcbpc_intra(bs);            /*解码得到mcbpc*/
        mb->mode = mcbpc & 7;                   /*宏块编码模式*/
        cbpc = (mcbpc >> 4);                    /*组合得到cbpc*/
        acpred_flag = BitstreamGetBit(bs);      /*AC预测方向*/
        cbpy = get_cbpy(bs, 1);                 /*解码得到cbpy*/
        cbp = (cbpy << 2) | cbpc;               /*组合得到cbp*/
        mb->quant = quant;
        mb->mvs[0].x = mb->mvs[0].y = mb->mvs[1].x = mb->mvs[1].y =
        mb->mvs[2].x = mb->mvs[2].y = mb->mvs[3].x = mb->mvs[3].y =0;
        decoder_mbintra(dec, mb, x, y, acpred_flag, cbp, bs, quant, intra_dc_
threshold, 0);
      }
    }
```

上述代码实现 I 帧图像的解码，首先从码流中获取 mcbpc、AC 预测方向、cbpy，将运动量清零，调用核心函数 decoder_mbintra 进行解码，以宏块为单位循环处理。

② decoder_mbintra 对 Intra 宏块做正真的解码

该函数是 Intra 块的核心解码函数，进行熵解码、反量化、反变换，最后把解码后的宏块数据更新到当前解码图像空间。

Intra 块的解码 decoder_mbintra()的流程如图 14-8 所示。

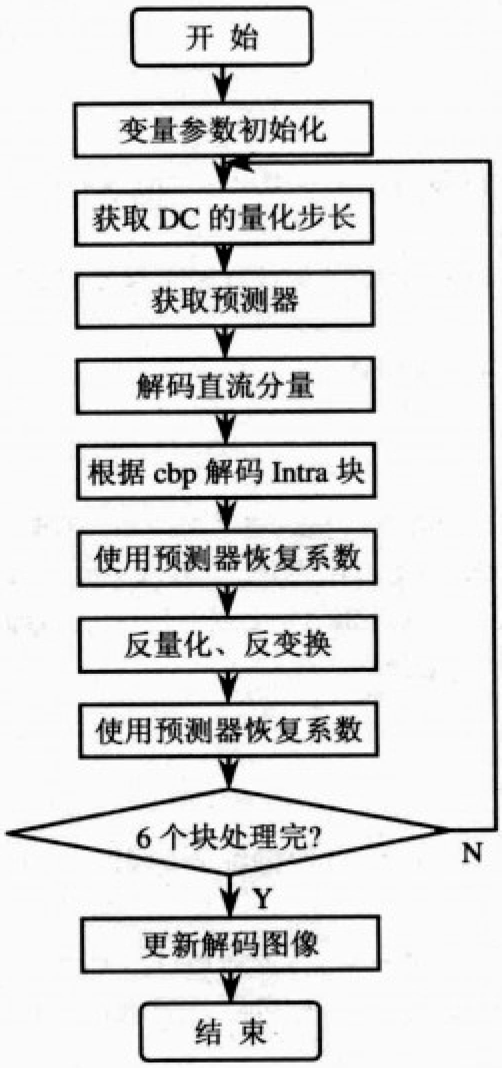


图 14-8 decoder_mbintra()流程图

根据该流程，代码实现如下。

```
/* decode an intra macroblock */
static void decoder_mbintra(DECODER * dec, MACROBLOCK * pMB,
        const uint32_t x_pos, const uint32_t y_pos, const uint32_t acpred_flag,
const uint32_t cbp,
        Bitstream * bs, const uint32_t quant, const uint32_t intra_dc_threshold,
const unsigned int bound)
{
  DECLARE_ALIGNED_MATRIX(block, 6, 64, int16_t, CACHE_LINE);  /*DCT系数*/
  DECLARE_ALIGNED_MATRIX(data, 6, 64, int16_t, CACHE_LINE);   /*图像数据*/
  uint32_t stride = dec->edged_width;             /*图像边做扩展的图像Y宽度*/
  uint32_t stride2 = stride / 2;                  /*图像边做扩展的图像U宽度*/
  uint32_t next_block = stride * 8;               /*修改指向块的横跨宽度*/
  uint32_t i;
  uint32_t iQuant = pMB->quant;
  uint8_t *pY_Cur, *pU_Cur, *pV_Cur;
  /*获取指向图像空间的指针*/
  pY_Cur = dec->cur.y + (y_pos << 4) * stride + (x_pos << 4);
  pU_Cur = dec->cur.u + (y_pos << 3) * stride2 + (x_pos << 3);
  pV_Cur = dec->cur.v + (y_pos << 3) * stride2 + (x_pos << 3);
  memset(block, 0, 6 * 64 * sizeof(int16_t)); /* 图像DCT系数清空 */
  /*循环对6个块处理*/
  for (i = 0; i < 6; i++) {
   uint32_t iDcScaler = get_dc_scaler(iQuant, i < 4); /*获取DC分量的量化步长*/
   int16_t predictors[8];                       /*预测器*/
   int start_coeff;
   /*预测DCT系数，得到预测器*/
  predict_acdc(dec->mbs,x_pos,y_pos,dec->mb_width,i,&block[i  *  64],iQuant,
iDcScaler, predictors, bound);
   if (!acpred_flag)  pMB->acpred_directions[i] = 0;
   /*得到DCT的直流分量*/
   if (quant < intra_dc_threshold) {
    int dc_size;
    int dc_dif;
    dc_size = i < 4 ? get_dc_size_lum(bs) : get_dc_size_chrom(bs);
    dc_dif = dc_size ? get_dc_dif(bs, dc_size) : 0;
    if (dc_size > 8) BitstreamSkip(bs, 1); /* marker */
    block[i * 64 + 0] = dc_dif;
    start_coeff = 1;
 } else
   start_coeff = 0;
   if (cbp & (1 << (5 - i))) /* 宏块做编码标识*/
   {
    int direction = dec->alternate_vertical_scan ? 2 : pMB->acpred_directions[i];
    get_intra_block(bs, &block[i * 64], direction, start_coeff);
 /*获取当前块的系数*/
 }
```

```
    /*使用预测器，恢复编码前系数值*/
        add_acdc(pMB, i, &block[i * 64], iDcScaler, predictors, dec->bs_version);
        /*H.263反量化*/
        dequant_h263_intra(&data[i  *  64],  &block[i  *  64],  iQuant,  iDcScaler,
dec->mpeg_quant_matrices);
        /*反DCT变换*/
        idct((short * const)&data[i * 64]);
      }/*循环对6个块处理*/
      /*把解码后的宏块图像复制到当前解码图像空间*/
      transfer_16to8copy(pY_Cur, &data[0 * 64], stride);
      transfer_16to8copy(pY_Cur + 8, &data[1 * 64], stride);
      transfer_16to8copy(pY_Cur + next_block, &data[2 * 64], stride);
      transfer_16to8copy(pY_Cur + 8 + next_block, &data[3 * 64], stride);
      transfer_16to8copy(pU_Cur, &data[4 * 64], stride2);
      transfer_16to8copy(pV_Cur, &data[5 * 64], stride2);
    }
```

上述代码实现 Intra 块的解码工作，首先初始化必要的参数和变量，解码得到 DC 直流分量的量化步长，接着预测 DCT 系数以获得预测器，解码直流分量、交流分量，反量化、反变换。最后把解码的宏块数据更新到当前解码图像空间。

③ decoder_pframe

P 帧是主要的编码类型，解码器可以根据编码器的功能支持把多余的代码屏蔽。XviD 支持的解码器较复杂，如 B 帧解码、GMC 等，这里主要讨论解码 SP 档级的码流。

P 帧的解码 decoder_pframe()的流程如图 14-9 所示。

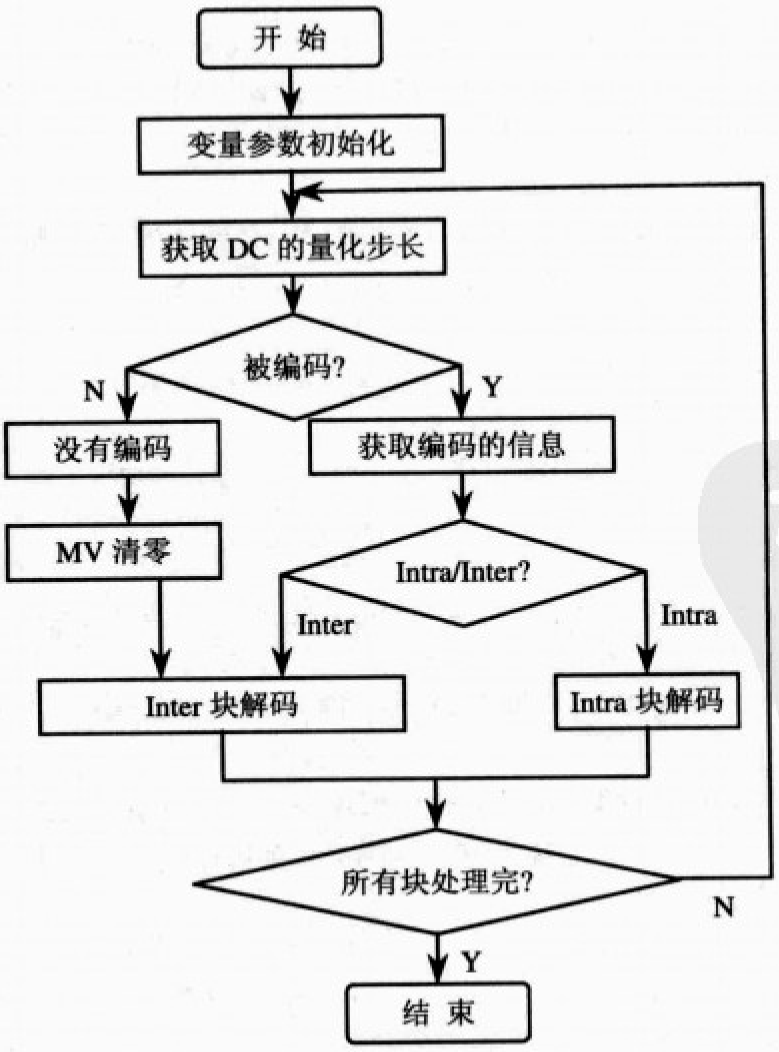


图 14-9　decoder_pframe()流程图

根据该流程，代码实现如下。

```
    void decoder_pframe(DECODER * dec,   Bitstream * bs, int rounding,int reduced_
resolution,
                int quant,int fcode,int intra_dc_threshold)
    {
    uint32_t x, y;
    uint32_t bound;
    VECTOR  mv;
    const VECTOR zerovec={0,0};
    uint32_t mb_width = dec->mb_width;
    uint32_t mb_height = dec->mb_height;
    uint32_t width = dec->width;
    uint32_t edged_width = dec->edged_width;
        /*循环所有宏块*/
        for (y=0; y<mb_height; y++) {
        for (x = 0; x < mb_width; x++) {
            MACROBLOCK *mb;
            mb = &dec->mbs[y * dec->mb_width + x];
            if (!(BitstreamGetBit(bs)))                          /*当前宏块已经被编码*/
            {
                uint32_t mcbpc, cbpc, cbpy, cbp;
                uint32_t intra, acpred_flag = 0;
                mcbpc = get_mcbpc_inter(bs);                  /*解码得到mcbpc*/
                mb->mode = mcbpc & 7;
                cbpc = (mcbpc >> 4);                          /*组合得到cbpc*/
                intra = (mb->mode == MODE_INTRA);             /*是否Intra宏块*/
                if (intra)  acpred_flag = BitstreamGetBit(bs);  /*Intra块的AC预测*/
                cbpy = get_cbpy_asm_dm642(bs,intra);          /*解码得到cbpy*/
                cbp = (cbpy << 2) | cbpc;                     /*组合得到CBP*/
                mb->quant = quant;                            /*量化初始化*/
                if (mb->mode == MODE_INTER){ /*编码模式是MODE_INTER*/
                    get_motion_vector(dec, bs, x, y, 0,&mv, fcode, bound);
                                                              /*解码得到MV.x , MV.y*/
                    mb->mvs[0] = mb->mvs[1] = mb->mvs[2] = mb->mvs[3] = mv;
                                                              /*初始化MVS*/
                    decoder_mbinter(dec, mb, x, y, cbp, bs,  mv, 0,x);
                                                              /*Inter 解码*/
                } else {                                      /*编码模式是MODE_INTRA */
                    mb->mvs[0].x = mb->mvs[1].x = mb->mvs[2].x = mb->mvs[3].x =
0;/*MVS置0 */

                    mb->mvs[0].y = mb->mvs[1].y = mb->mvs[2].y = mb->mvs[3].y = 0;/*
MVS置0*/

                    decoder_mbintra(dec, mb, x, y, acpred_flag, cbp,/*解码Intra块*/
                            bs, quant, intra_dc_threshold, bound);
                    continue;
                }
            } else {                                     /*当前宏块没有被编码not_coded */
                mb->mode = MODE_NOT_CODED;
                mb->quant = quant;
```

```
                mb->mvs[0].x = mb->mvs[1].x = mb->mvs[2].x = mb->mvs[3].x = 0;/* MVS
置0*/
                mb->mvs[0].y = mb->mvs[1].y = mb->mvs[2].y = mb->mvs[3].y = 0;/* MVS
置0*/
                decoder_mbinter(dec, mb, x, y, 0, bs, zerovec, 0, x);/* Inter解码*/
            }
        }   //x 循环
    }   //y 循环
}
```

上述代码实现 P 帧的解码，根据前面编码的流程知道，宏块有 3 种编码模式：Inter、Intra 和 not_coded。而对于 not_coded 同样是属于 Inter 块的编码范畴，在解码中同样要作运动补偿，只是不再做实质的解码工作。

④ decoder_mbinter 对 Inter 块实现解码

Inter 块解码支持 1/2 像素精度，Inter 块的 decoder_mbinter()的流程如图 14-10 所示。

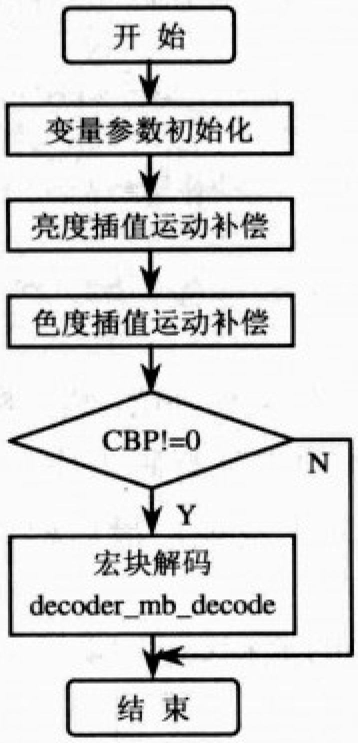


图 14-10 decoder_mbinter ()流程图

根据该流程，代码实现如下。

```
void decoder_mbinter(DECODER * dec, const MACROBLOCK * pMB,
        const uint32_t x_pos, const uint32_t y_pos, const uint32_t cbp, Bitstream
* bs,
        const uint32_t rounding, const int ref, const int bvop)
{
  uint32_t stride = dec->edged_width;
  uint32_t stride2 = stride / 2;
  uint32_t i;
  uint8_t *pY_Cur, *pU_Cur, *pV_Cur;
  int uv_dx, uv_dy;
  VECTOR mv[4]; /* local copy of mvs */
  /*获取指向图像空间的指针*/
  pY_Cur = dec->cur.y + (y_pos << 4) * stride + (x_pos << 4);
  pU_Cur = dec->cur.u + (y_pos << 3) * stride2 + (x_pos << 3);
```

```
    pV_Cur = dec->cur.v + (y_pos << 3) * stride2 + (x_pos << 3);
    for (i = 0; i < 4; i++)  mv[i] = pMB->mvs[i];   /*获取运动向量*/
    validate_vector(mv, x_pos, y_pos, dec);          /*检查运动向量的合法性*/
    /*****************************运动补偿******************************/
    if ((pMB->mode != MODE_INTER4V))
    { /* 编码模式是INTER、INTER_Q、NOT_CODED、FORWARD、BACKWARD */
     uv_dx = mv[0].x;
     uv_dy = mv[0].y;
     uv_dx = (uv_dx >> 1) + roundtab_79[uv_dx & 0x3];
  uv_dy = (uv_dy >> 1) + roundtab_79[uv_dy & 0x3];
  /*亮度插值*/
  interpolate16x16_switch(dec->cur.y,dec->refn[ref].y,16*x_pos,16*y_pos,mv[0].
x, mv[0].y, stride, rounding);
    }
    /*色度插值 */
    interpolate8x8_switch(dec->cur.u, dec->refn[ref].u, 8 * x_pos, 8 * y_pos,uv_dx,
uv_dy, stride2, rounding);
    interpolate8x8_switch(dec->cur.v, dec->refn[ref].v, 8 * x_pos, 8 * y_pos,uv_dx,
uv_dy, stride2, rounding);
  /*****************************运动补偿******************************/
    /*根据cbp*/
    if (cbp) decoder_mb_decode(dec, cbp, bs, pY_Cur, pU_Cur, pV_Cur, pMB);
  }
```

上述代码实现 Inter 块的解码，首先作运动补偿，然后根据 cbp 编码模式决定是否做宏块解码。

⑥ decoder_mb_decode 实现宏块的实际解码工作

首先 VLD 熵解码、反量化和 IDCT 变换，最后根据 CBP 编码模式决定块是否更新到当前解码帧图像。

Inter 块的 decoder_ mb_decode ()的流程如图 14-11 所示。

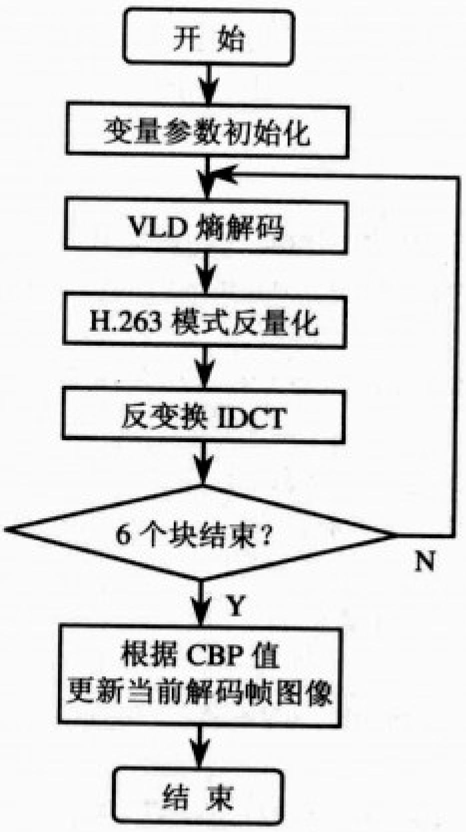


图 14-11　decoder_mb_decode()流程图

根据该流程，代码实现如下。

```
void decoder_mb_decode(DECODER * dec, const uint32_t cbp, Bitstream * bs,
                uint8_t * pY_Cur, uint8_t * pU_Cur, uint8_t * pV_Cur,
                const int reduced_resolution,const MACROBLOCK * pMB)
{
DECLARE_ALIGNED_MATRIX(block, 1, 64, int16_t, CACHE_LINE);    /*系数*/
DECLARE_ALIGNED_MATRIX(data, 6, 64, int16_t, CACHE_LINE);     /*图像*/
int stride = dec->edged_width;                /*图像边扩展后的宽度*/
int next_block = stride * 8;                  /*亮度的下一个块*/
const int stride2 = stride/2;                 /*色度的下一个快*/
int i;
const uint32_t iQuant = pMB->quant;   /*量化步长*/
const int direction = dec->alternate_vertical_scan ? 2 : 0;   /*AC扫描方式*/
for (i = 0; i < 6; i++) {
    if (cbp & (1 << (5 - i))) {                           /*块做了编码*/
        memset(block, 0, 64 * sizeof(int16_t));           /*清零*/
        get_inter_block(bs, block, direction);            /*VLD熵解码*/
        dequant_h263_inter (&data[i * 64], block, iQuant, NULL);/*H263反量化*/
        idct(&data[i * 64]);                              /*IDCT变换*/
    }
}
/*根据CBP值决定是否加到当前图像*/
if (cbp & 32) transfer_16to8add(pY_Cur, &data[0 * 64], stride);
if (cbp & 16) transfer_16to8add(pY_Cur + 8, &data[1 * 64], stride);
if (cbp & 8)  transfer_16to8add(pY_Cur + next_block, &data[2 * 64], stride);
if (cbp & 4)  transfer_16to8add(pY_Cur + 8 + next_block, &data[3 * 64], stride);
if (cbp & 2)  transfer_16to8add(pU_Cur, &data[4 * 64], stride2);
if (cbp & 1)  transfer_16to8add(pV_Cur, &data[5 * 64], stride2);
}
```

上述代码实现 Inter 块的解码工作，根据 cbp 值决定是否做解码。首先解码宏块空间 block 清零，VLD 熵解码，然后反量化、反变换。最后根据 cbp 值决定是否更新当前解码帧图像。

为了提高解码器解码速度，解码器的最底层模块 get_inter_block、dequant_h263_inter、idct、transfer_16to8add 等，要使用平台的汇编来优化和设计。下一节将介绍这些核心模块的 MMX/SSE/DM642 汇编优化。

3）销毁解码器实例

对应创建编码器工作，释放申请的所有内存，删除解码器句柄。

```
int decoder_destroy(DECODER * dec)
{
xvid_free(dec->last_mbs);                  //释放宏块结构体
xvid_free(dec->mbs);                       //释放宏块结构体
image_destroy(&dec->refn[0], dec->edged_width, dec->edged_height);
//释放参考帧0的空间
```

```
    image_destroy(&dec->refn[1], dec->edged_width, dec->edged_height);
    //释放参考帧1的空间
    image_destroy(&dec->tmp, dec->edged_width, dec->edged_height);
    //释放临时帧的空间
    image_destroy(&dec->cur, dec->edged_width, dec->edged_height);
    //释放当前帧的空间
    xvid_free(dec);                         //释放、删除解码器
    return 0;
    }
```

XviD 的 MPEG-4 视频编/解码算法结构清晰、定点算法，非常适合硬件平台来实现。可以说 XviD 是目前 MPEG-4 算法开源代码中最优秀的工程实现。一方面 XviD 已经针对主流的 CPU 做了汇编优化，另一方面为其他平台移植提供了技术方向。如表 14-1 所示是 XviD MPEG-4 视频编/解码算法的底层模块。

表 14-1　XviD MPEG-4 视频 CODEC 底层模块

核心模块	编码	解码	功能描述
transfer_8to16copy	×		图像精度扩展，8～16 位
transfer_16to8copy	×	×	图像像素嵌位 8 位（0～255）
transfer_8to16sub	×		图像块相减，并扩展精度
transfer_16to8add	×	×	差值加到图像，并嵌位到 8 位
transfer8x8_copy	×		块复制
quant_h263_intra	×		Intra 块的 h263 方式量化
quant_h263_inter	×		Intrer 块的 h263 方式量化
dequant_h263_intra	×	×	Intra 块的 h263 方式反量化
dequant_h263_inter	×	×	Intrer 块的 h263 方式反量化
Fdct	×		前向 DCT
Idct	×	×	反向 DCT
interpolate8x8_halfpel_h	×	×	水平 1/2 像素差值
interpolate8x8_halfpel_v	×	×	垂直 1/2 像素差值
interpolate8x8_halfpel_hv	×	×	斜角 1/2 像素差值
sad16	×		计算图像宏块的 SAD
sad8	×		计算图像块的 SAD
dev16	×		计算图像宏块的平均值的偏差
calc_cbp	×		计算 6 个块的 CBP

从表 14-1 分析可以清晰地看出，编码器的工作包含了解码器的任务。为提高 CODEC 的执行效率，这些底层模块必须进行汇编优化，下面就介绍底层模块的优化方法。

14.2.2 MMX/SSE 2实现XviD CODEC

上一节对 XviD CODEC 的系统框架做了功能剖析和技术说明，现在开始对最底层的模块如 DCT/IDCT、Q/IQ、VLC/VLD、SAD 等作汇编优化和设计。第 12 章已经对 MMX/SSE 2 汇编编程、汇编指令结构等作了详细介绍。基于通用 CPU 进行多媒体编程，汇编语言提供了数据打包、单指令多数据处理等特点的高性能支持。

由于算法系统是 C 语言编程，而在使用汇编优化核心模块的时候，就涉及了混合编程技术。汇编编写的函数要被 C 语言调用，则在编程时要遵守一定的规则。如在汇编函数名称前加下画线“_”。

1．MMX 汇编指令优化核心模块

XviD CODEC 的汇编程序支持 NASM 汇编器，它是一个为可移植性与模块化而设计的 80×86 的汇编器。它支持相当多的目标文件格式，包括 Linux、NetBSD/FreeBSD、a.out、ELF、COFF、微软 16 位的 OBJ 和 Win32。它还可以输出纯二进制文件。它的语法设计得相当地简洁易懂，和 Intel 语法相似但更简单。它支持 Pentium、P6、MMX、3DNow!、SSE 和 SSE 2 指令集。

1）一般宏定义

下面的宏定义中使用了宏汇编中表达式操作符%，它把后面的文本解释为表达式，功能是取表达式的最终的值。使用这个操作符可以把表达式的值作为实参引用，而不是引用表达式文本本身。

%macro 表示定义多行宏，%endmacro 表示多行宏的结束。在%macro 一行上宏名后面的数字 1 定义了宏可以接收的参数的个数，宏定义里面的%1 用来引用宏调用中的第一个参数。对于一个有多个参数的宏，参数序列可以这样写：%2、%3 等。%define 用来定义单行宏，NASM 的详细开发技术请参考 Nasm 中文手册。

```
BITS 32                                     ; 表明32位指令
%macro cglobal 1                            ; cglobal宏定义
%ifdef PREFIX
    %ifdef MARK_FUNCS
        global _%1:function %1.endfunc-%1
        %define %1 _%1:function %1.endfunc-%1
    %else
        global _%1                          ;有效定义，定义全局函数
        %define %1 _%1                      ;在汇编中，使用没有“_”的函数名称
    %endif
%else
    %ifdef MARK_FUNCS
        global %1:function %1.endfunc-%1
    %else
        global %1
    %endif
```

```
%endif
%endmacro
```

上面使用宏定义了函数的名称，宏名为 cglobal，由于在汇编文件的汇编选项设置中定义了 PREFIX，而没有定义 MARK_FUNCS，所以宏的实际定义为上述代码中加黑显示的内容，例如：

```
global _myfunc
define myfunc _myfunc
```

2）只读段.rodata 存放常量

```
;=========================================================================
; 只读数据段(Read Only Data)
;=========================================================================
%ifdef FORMAT_COFF
SECTION .rodata
%else
SECTION .rodata align=16   ;只读数据段，16字节（128位）对齐
%endif
ALIGN 16               ;16字节对齐
mmx_one:
dw 1, 1, 1, 1          ;定义的字
```

上述代码定义了 16 字节对齐的常量 mmx_one，4 个常量中每个常量长度为 16 比特，数据放置在只读段.rodata。

3）C 语言函数 transfer_8to16copy_c 的功能

C 语言函数 transfer_8to16copy_c 的功能是把像素值从 8 位扩展到 16 位。

```
void
transfer_8to16copy_c(int16_t * const dst, const uint8_t * const src, uint32_t
stride)
{
uint32_t i, j;
for (j = 0; j < 8; j++) {
    for (i = 0; i < 8; i++) {
        dst[j * 8 + i] = (int16_t) src[j * stride + i];
    }
}
}
```

上述代码有两层循环，每次处理一个单元，即把 8 位的像素值扩展为 16 位。

MMX 的寄存器是 64 位，这样 MMX 指令可以一次处理 8 个字节。该函数的汇编优化是 transfer_8to16copy_mmx()，该函数优化的思路是把要复制的数据打包形成 64 位，每次存储两行，执行 4 次宏处理，下面是汇编优化的结果。

```
SECTION .text                     ;.text 段
cglobal transfer_8to16copy_mmx    ;函数声明：_ transfer_8to16copy_mmx
%macro COPY_8_TO_16 1             ;定义宏COPY_8_TO_16开始
  movq mm0, [eax]                 ;取64bit，第一行8个像素点
```

```
  movq mm1, [eax+edx]                ;取64bit，第二行8个像素点
  movq mm2, mm0                      ;mm2=mm0
  movq mm3, mm1                      ;mm3=mm1
  punpcklbw mm0, mm7                 ;mm0的低4Byte扩展成word
  movq [ecx+%1*32], mm0              ;mm0的内容存到[ecx+%1*32]
  punpcklbw mm1, mm7                 ;mm1的低4Byte扩展成word
  movq [ecx+%1*32+16], mm1           ;mm1的内容存到[ecx+%1*32+16]
  punpckhbw mm2, mm7                 ;mm2的低4Byte扩展成word
  punpckhbw mm3, mm7                 ;mm3的低4Byte扩展成word
  lea eax, [eax+2*edx]               ;修改eax值，指向当前行的第三行
  movq [ecx+%1*32+8], mm2            ;mm2的内容存到[ecx+%1*32+8]
  movq [ecx+%1*32+24], mm3           ;mm3的内容存到[ecx+%1*32+24]
%endmacro                            ;定义宏COPY_8_TO_16结束
ALIGN 16                             ;下面地址是16字节对齐
transfer_8to16copy_mmx:              ;实际是_transfer_8to16copy_mmx，前面有define定义。
  mov ecx, [esp+ 4] ; Dst            ;取第一个参数 dst
  mov eax, [esp+ 8] ; Src            ;取第二个参数 src
  mov edx, [esp+12] ; Stride         ;取第三个参数 stride
  pxor mm7, mm7                      ;mm7清零
  COPY_8_TO_16 0                     ;第一次展开宏，处理0、1行
  COPY_8_TO_16 1                     ;第二次展开宏，处理2、3行
  COPY_8_TO_16 2                     ;第三次展开宏，处理4、5行
  COPY_8_TO_16 3                     ;第四次展开宏，处理6、7行
  ret                                ;函数返回
.endfunc
```

经过分析，上述汇编函数的显示定义如下，可以看出汇编函数名称前必须加“_”，以保证能够被 C 语言调用。传入的参数在堆栈中，使用 ESP 指针加偏移量读取。

```
;-----------------------------------------------------------------------
; void transfer_16to8copy_mmx(uint8_t * const dst,
;                             const int16_t * const src,
;                             uint32_t stride);
;-----------------------------------------------------------------------
  SECTION .text
  global _transfer_8to16copy_mmx
ALIGN 16
_transfer_8to16copy_mmx:
  ;汇编指令
  ;汇编指令
  ret
.endfunc
```

上述代码是函数 transfer_16to8copy_mmx 的汇编实现。汇编中函数名称前加了下画线“_”。使用 global 关键字声明该函数为全局函数，使得 C 语言能调用该函数。

其他有的核心模块也使用了 MMX 汇编指令作优化。开发过程与 transfer_8to16copy_mmx 类似。详情参考 xx_mmx.asm 文件，这些模块的优化使用了 MMX 指令。

2. SSE 2 汇编指令优化核心模块

还有一部分核心模块使用了 SSE 2 汇编指令作优化。SSE 2 汇编语言的数据寄存器是 128 位，共 8 个，即 XMM0～XMM7。这样 SSE 指令可以一次处理 16 字节，相比 MMX 指令的一次处理 8 字节，效率提高了一倍。

MMX 和 SSE 2 的一般宏定义相同，常量同样放在.rodata 段下，地址 16 字节对齐。

下面以 sad 模块的 SSE 2 汇编优化为案例，分析 SSE 2 汇编指令优化过程，优化思路是每次处理宏块的 2 行，执行 8 次宏处理。

sad16 的 C 语言实现如下。

```
uint32_t sad16_c(const uint8_t * const cur, const uint8_t * const ref,
    const uint32_t stride,  const uint32_t best_sad)
{
uint32_t sad = 0;
uint32_t i, j;
uint8_t const *ptr_cur = cur;    //当前帧指针
uint8_t const *ptr_ref = ref;    //参考帧指针
for (j = 0; j < 16; j++) {       //外循环
    for (i = 0; i < 16; i++) {   //内循环
        int pixel = (ptr_cur[i] - ptr_ref2[i]);  //对应位置相间
        sad += abs(pixel);       //累加绝对值
    }
    ptr_cur += stride;           //指向下一行
    ptr_ref += stride;           //指向下一行
}
return sad;                      //返回sad值
}
```

上述程序是计算对应块的 sad 值，两层循环，每次只计算一个像素点的差值，先采用 SSE 的指令优化该函数。下面代码是该函数的 SSE 优化结果。

```
;------------------------------------------------------------------------
; uint32_t sad16_sse2 (const uint8_t * const cur, <- assumed aligned!
;                     const uint8_t * const ref,
;                      const uint32_t stride,
;                     const uint32_t /*ignored*/);
;------------------------------------------------------------------------
%macro SAD_16x16_SSE2 0        ;定义宏
 movdqu  xmm0, [edx]           ;为16字节对齐读取，=>xmm0
 movdqu  xmm1, [edx+ecx]       ;为16字节对齐读取，=>xmm1
 lea edx,[edx+2*ecx]           ;修改edx
 movdqa  xmm2, [eax]           ;16字节对齐读取，=>xmm2
 movdqa  xmm3, [eax+ecx]       ;16字节对齐读取，=>xmm3
 lea eax,[eax+2*ecx]           ;修改eax
 psadbw  xmm0, xmm2            ;8字节的sad值，结果放置在低16位
 paddusw xmm6,xmm0             ;8个无符号16位饱和相加
```

```
  psadbw  xmm1, xmm3              ;8个字节的sad值，结果放置在低16位
  paddusw xmm6,xmm1               ;8个无符号16位饱和相加
%endmacro                         ;宏定义结束
ALIGN 16                          ;16字节对齐
sad16_sse2:                       ;函数声明，_sad16_see2
  mov eax, [esp+ 4] ; cur (assumed aligned)   ;取第一个参数，cur是16字节对齐
  mov edx, [esp+ 8] ; ref     ;取第二个参数，ref对齐不限制
  mov ecx, [esp+12] ; stride      ;取第三个参数
  pxor xmm6, xmm6 ; accum         ;xmm6 = 0
  SAD_16x16_SSE2                  ;宏展开，处理0、1行
  SAD_16x16_SSE2                  ;宏展开，处理2、3行
  SAD_16x16_SSE2                  ;宏展开，处理4、5行
  SAD_16x16_SSE2                  ;宏展开，处理6、7行
  SAD_16x16_SSE2                  ;宏展开，处理8、9行
  SAD_16x16_SSE2                  ;宏展开，处理10、11行
  SAD_16x16_SSE2                  ;宏展开，处理12、13行
  SAD_16x16_SSE2                  ;宏展开，处理14、15行
  pshufd  xmm5, xmm6, 00000010b   ;根据00000010b重排xmm6
  paddusw xmm6, xmm5              ;8个无符号16位饱和相加
  pextrw  eax, xmm6, 0            ;扩展字，前面补零
  ret
.endfunc                          ;函数声明结束
```

在上述代码中，宏 SAD_16x16_SSE 2 处理 16×16 宏块的 2 行，16 个的单字节的像素存储 SSE 的一个 128 位寄存器 XMM 中。调用 8 次宏，完成整个宏块的处理。

不论是 MMX 还是 SSE 2 汇编指令，优化的根本思路都是数据打包，单指令处理多数据源（SIMD）。

14.2.3 DM642下XviD CODEC优化

第 13 章曾介绍过 DM642 实现视频编/解码技术的基本过程。C 语言实现算法系统框架，线性汇编优化核心函数使模块尽可能软件流水，对于图像数据使用 DMA 双缓冲的方法，在 L2 的 SRAM 内存区开辟图像工作空间，后台开启 DMA 传输，配置 L2 Cache，增加 CPU 存取数据的命中率。使用 CCS 开发软件的优化工具，优化组合编译开关，辅助开发人员进行算法优化。

表 14-1 所示的 XviD MPEG-4 视频 CODEC 底层模块都需要汇编优化，或者 Intrinsic 指令优化。TI 的图像算法库 img64x.lib 提供了图像处理的常用算法函数，有 C 语言、线性汇编语言和并行汇编的源代码。一方面开发人员可以直接使用这些库函数，另一方面参考、模仿其汇编优化过程和编程思路，对自己算法逐步深入汇编优化，如图像变换 DCT，库 img64x.lib 就提供了并行度极高、完全建立了软件流水的函数 IMG_fdct_8x8()，运动估计 ME 中判断块最佳匹配的 SAD 值函数 IMG_sad_16x16()、IMG_sad_8x8()。在使用该库的时候，需特别注意输入源数据、输出结果的格式。如 DCT 中，输出的结果是每个元素左移了 4 位；计算 SAD 时，注意两个图像块边界是否和地址对齐，为了提高并行度尽可能使用对齐的数据存取指令。

下面对编/解码中的运动补偿 MC 模块作 DM642 优化。运动补偿的原理是根据运动向量 MV 的值，对当前宏块补偿出新的宏块，补偿的方式根据像素的精度有 1/2、1/4、1/8 之分。现在对 1/2 精度的函数作优化，其他精度的补偿优化读者可通过参考此例来实现。

1．水平方向插值补偿

水平方向插值的原理是图像块的第 0 列、第 1 列的元素对应取平均，结果放置在新块的第 0 列，图像块的第 1 列、第 2 列的元素对应取平均，结果放置新块的第 1 列，依次类推，直到处理完整个图像块。新块即为水平插值补偿块，C 语言代码如下。

```
void interpolate8x8_halfpel_h(uint8_t* const dst,const uint8_t* const src,
                    const uint32_t stride, const uint32_t rounding)
{
uint32_t j;
int r = 1 - rounding;     /*舍尾方式*/
for (j = 0; j < 8*stride; j+=stride){                            // 每次循环处理一列
    dst[j + 0] = (uint8_t)((src[j + 0] + src[j + 1] + r )>>1);  //0元素和1元素取平均
    dst[j + 1] = (uint8_t)((src[j + 1] + src[j + 2] + r )>>1);  //1元素和2元素取平均
    dst[j + 2] = (uint8_t)((src[j + 2] + src[j + 3] + r )>>1);  //2元素和3元素取平均
    dst[j + 3] = (uint8_t)((src[j + 3] + src[j + 4] + r )>>1);  //3元素和4元素取平均
    dst[j + 4] = (uint8_t)((src[j + 4] + src[j + 5] + r )>>1);  //4元素和5元素取平均
    dst[j + 5] = (uint8_t)((src[j + 5] + src[j + 6] + r )>>1);  //5元素和6元素取平均
    dst[j + 6] = (uint8_t)((src[j + 6] + src[j + 7] + r )>>1);  //6元素和7元素取平均
    dst[j + 7] = (uint8_t)((src[j + 7] + src[j + 8] + r )>>1);  //7元素和8元素取平均
}
}
```

此函数处理数据规整，源和目标存储区没有依赖性。源和目标地址均为图像空间，源地址不对齐，目标地址对齐。线性汇编优化如下。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;               interpolate8x8_halfpel_h_asm_dm642                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    .sect ".text:_interpolate8x8_halfpel_h_asm_dm642"
        .global _interpolate8x8_halfpel_h_asm_dm642
_interpolate8x8_halfpel_h_asm_dm642  .cproc A_dst0, B_src0, A_stride,B_stride
.no_mdep            ;存储区没有依赖性，这样便于流水
.reg     B_src1_FIX_F, B_src1_FEDC:B_src1_BA98, B_src1_GFED, B_src1_CBA9 ;定义
的寄存器
.reg     A_src0_FIX_7, A_src0_7654:A_src0_3210, A_src0_8765, A_src0_4321
.reg     B_dst1_FEDC:B_dst1_BA98, A_dst0_7654:A_dst0_3210
.reg     B_dst1, A_src1, A_i, A_pitch,B_pitch
MVK 3,          A_i           ;循环次数
ADD B_src0,     B_stride,    A_src1   ;
ADD      A_dst0,      A_stride,      B_dst1   ;
SHR      A_stride,          2,        A_pitch
SHR      B_stride,          2,        B_pitch
mc8x8_hpel_h: .trip 4, 4, 4                                    ; 每次处理两行，循环4次
```

```
        LDNW  .D1T2    *+A_src1[2],          B_src1_FIX_F    ; 地址不对齐读取
        LDNDW .D1T2    *A_src1++[B_pitch], B_src1_FEDC:B_src1_BA98 ;
        ;;
    SHRMB    .2  B_src1_FIX_F,    B_src1_FEDC, B_src1_GFED ;右移并拼接: fix_F ||
(FEDC>>8)
    ;;      AVGU4 .2    B_src1_FEDC,B_src1_GFED, B_dst1_FEDC ; 4对字节对应取平均值
        ;;
        STDW  .D2T2    B_dst1_FEDC:B_dst1_BA98,      *B_dst1++[A_pitch] ; 存结果
        ;;
  [A_i] BDEC          mc8x8_hpel_h, A_i
        .return
        .endproc
```

上述代码首先使用地址不对齐指令读取数据源。然后右移并拼接形成 4 字节，接着取 4 对字节对应求平均，最后存储结果。每次处理两行，循环 4 次。

2. 垂直方向插值补偿

垂直方向插值的原理是图像块的第 0 行、第 1 行的元素对应取平均，结果放置在新块的第 0 行，图像块的第 1 行、第 2 行的元素对应取平均，结果放置新块的第 1 行，依次类推，直到处理完整个图像块，新块即为垂直插值补偿块。C 语言代码如下。

```
void interpolate8x8_halfpel_v(uint8_t* const dst,const uint8_t* const src,
                    const uint32_t stride, const uint32_t rounding)
{
uint32_t j;
int r = 1 - rounding;
for (j = 0; j < 8*stride; j+=stride)                //每次循环处理一行
{
    dst[j + 0] = (uint8_t)((src[j + 0] + src[j + stride + 0] + r )>>1);
    //0元素和次行0元素取平均
    dst[j + 1] = (uint8_t)((src[j + 1] + src[j + stride + 1] + r )>>1);
    //1元素和次行1元素取平均
    dst[j + 2] = (uint8_t)((src[j + 2] + src[j + stride + 2] + r )>>1);
    //2元素和次行2元素取平均
    dst[j + 3] = (uint8_t)((src[j + 3] + src[j + stride + 3] + r )>>1);
    //3元素和次行3元素取平均
    dst[j + 4] = (uint8_t)((src[j + 4] + src[j + stride + 4] + r )>>1);
    //4元素和次行4元素取平均
    dst[j + 5] = (uint8_t)((src[j + 5] + src[j + stride + 5] + r )>>1);
    //5元素和次行5元素取平均
    dst[j + 6] = (uint8_t)((src[j + 6] + src[j + stride + 6] + r )>>1);
    //6元素和次行6元素取平均
    dst[j + 7] = (uint8_t)((src[j + 7] + src[j + stride + 7] + r )>>1);
    //7元素和次行7元素取平均
}
}
```

此函数处理数据同样规整，存储区没有依赖性。源和目标地址均为图像空间，源地址不对齐，目标地址对齐。线性汇编优化如下。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;            interpolate8x8_halfpel_v_asm_dm642          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  .sect ".text:_interpolate8x8_halfpel_v_asm_dm642"
      .global _interpolate8x8_halfpel_v_asm_dm642
_interpolate8x8_halfpel_v_asm_dm642  .cproc A_dst0, B_src0, A_stride,B_stride
.no_mdep          ;存储区无依赖性
.regB_dst7_7654:B_dst7_3210, B_dst6_7654:B_dst6_3210
    ;;
    .regB_src8_7654:B_src8_3210, B_src7_7654:B_src7_3210
;;
.regA_src2_7654:A_src2_3210, A_src1_7654:A_src1_3210
.regA_src0_7654:A_src0_3210
.reg  B_dst1, A_src1, A_i, A_pitch,AB_stride
.regcst_32, B_pitch,BA_stride
SHR      B_stride,       3,          A_pitch
SHR      A_stride,       3,          B_pitch
SHL      B_stride,       2,          AB_stride
SHL      A_stride,       2,          BA_stride
ADD      B_src0,      AB_stride,   A_src1
ADD      A_dst0,      BA_stride,   B_dst1
    LDNDW   .D2T1     *B_src0++[A_pitch],  A_src0_7654:A_src0_3210   ;读取8字节
    LDNDW   .D2T1     *B_src0++[A_pitch],  A_src1_7654:A_src1_3210   ;读取8字节
    ;;
    AVGU4             A_src0_7654,     A_src1_7654,     A_dst0_7654 ;求4对字节平均
    AVGU4             A_src0_3210,     A_src1_3210,     A_dst0_3210 ;求4对字节平均
STDW     .D1T1    A_dst0_7654:A_dst0_3210,      *A_dst0++[B_pitch]    ;存储8字节
    ;;
   .return
   .endproc
```

上述代码中首先使用不对齐指令读取数据，然后求平均，最后存储结果。

3. 斜角插值补偿

斜角插值即对水平和垂直方向的4个像素求平均。或者理解为图像块先做水平插值，然后对插值后的新图像块做垂直插值，最后形成的新图像块即为斜角插值。C语言代码实现如下。

```
void interpolate8x8_halfpel_hv(uint8_t* const dst, const uint8_t* const src,
                     const uint32_t stride, const uint32_t rounding)
{
uint32_t j;
int r = 2- rounding;
for (j = 0; j < 8*stride; j+=stride)
```

```
{
    dst[j+0]=(uint8_t)((src[j+0]+src[j+1]+src[j+stride+0]+src[j+stride+1]+r)>>2);
    //4个像素求平均
    dst[j+1]=(uint8_t)((src[j+1]+src[j+2]+src[j+stride+1]+src[j+stride+2]+r)>>2);
    //4个像素求平均
    dst[j+2]=(uint8_t)((src[j+2]+src[j+3]+src[j+stride+2]+src[j+stride+3]+r)>>2);
    //4个像素求平均
    dst[j+3]=(uint8_t)((src[j+3]+src[j+4]+src[j+stride+3]+src[j+stride+4]+r)>>2);
    //4个像素求平均
    dst[j+4]=(uint8_t)((src[j+4]+src[j+5]+src[j+stride+4]+src[j+stride+5]+r)>>2);
    //4个像素求平均
    dst[j+5]=(uint8_t)((src[j+5]+src[j+6]+src[j+stride+5]+src[j+stride+6]+r)>>2);
    //4个像素求平均
    dst[j+6]=(uint8_t)((src[j+6]+src[j+7]+src[j+stride+6]+src[j+stride+7]+r)>>2);
    //4个像素求平均
    dst[j+7]=(uint8_t)((src[j+7]+src[j+8]+src[j+stride+7]+src[j+stride+8]+r)>>2);
    //4个像素求平均
}
}
```

斜角插值补偿主要还是求值平均。线性汇编优化过程如下。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;             interpolate8x8_halfpel_hv_asm_dm642          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   .sect ".text:_interpolate8x8_halfpel_hv_asm_dm642"
       .global _interpolate8x8_halfpel_hv_asm_dm642
_interpolate8x8_halfpel_hv_asm_dm642  .cproc A_dst0, B_src0, A_stride,B_stride
.no_mdep                    ;存储区无依赖性
.reg     A_dst1,B_src1,B_src2
.reg     A_i,A_pitch,B_pitch
.reg     A_1111,B_1111,A_22,B_22
.reg     A_7654:A_3210,A_8765:A_4321, B_7654:B_3210,B_8765:B_4321
.reg     C_7654:C_3210,C_8765:C_4321, D_7654:D_3210,E_7654:E_3210
.reg     A_0,A_1,A_2,A_3,A_4,A_5,A_6,A_7
.reg     C_1010,C_3232,C_5454,C_7676,C_2121,C_4343,C_6565,C_8787
.reg     A_76,A_54,A_32,A_10,B_76,B_54,B_32,B_10
.regC_76,C_54,C_32,C_10,D_76,D_54,D_32,D_10
.regE_76,E_54,E_32,E_10,F_76,F_54,F_32,F_10
ADD B_src0, B_stride,   B_src1
ADD B_src1, B_stride,   B_src2
ADD A_dst0, A_stride,   A_dst1
SHR B_stride,   2,  B_pitch
SHR A_stride,   2,  A_pitch
MVKL    0x01010101, A_1111          ;用来点积，实质是相加
MVKH    0x01010101, A_1111
MVKL    0x00020002, A_22            ;用来右移
MVKH    0x00020002, A_22
```

```
MV      A_22,        B_22
MV      A_1111,      B_1111
MVK 3,  A_i
LOOP_HV_88: .trip 4,4,4
LDNDW   *+B_src2(1),         C_8765:C_4321  ;地址不对齐读取8字节数据
LDNDW   *B_src2++[B_pitch],  C_7654:C_3210
LDNDW   *+B_src1(1),         B_8765:B_4321
LDNDW   *B_src1++[B_pitch],  B_7654:B_3210
LDNDW   *+B_src0(1),         A_8765:A_4321
LDNDW   *B_src0++[B_pitch],  A_7654:A_3210
PACK2   A_3210, B_3210, C_1010   ;打包低16位
PACKH2  B_3210, A_3210, C_3232   ;打包高16位
PACK2   A_7654, B_7654, C_5454
PACKH2  B_7654, A_7654, C_7676
PACK2   A_4321, B_4321, C_2121
PACKH2  B_4321, A_4321, C_4343
PACK2   A_8765, B_8765, C_6565
PACKH2  B_8765, A_8765, C_8787
    ;;
DOTPU4  C_1010, A_1111, A_0       ;与1点积，即4个元素相加
DOTPU4  C_2121, A_1111, A_1
DOTPU4  C_3232, A_1111, A_2
DOTPU4  C_4343, A_1111, A_3
DOTPU4  C_5454, A_1111, A_4
DOTPU4  C_6565, A_1111, A_5
DOTPU4  C_7676, A_1111, A_6
DOTPU4  C_8787, A_1111, A_7
    ;;
PACK2   A_1,A_0,A_10              ;低16位打包
PACK2   A_3,A_2,A_32
PACK2   A_5,A_4,A_54
PACK2   A_7,A_6,A_76
    ;;
ADD2    A_22,   A_10,   C_10      ;两个16位数分别加2
ADD2    A_22,   A_32,   C_32
ADD2    A_22,   A_54,   C_54
ADD2    A_22,   A_76,   C_76
;;
SHRU2   C_10,   A_22,   E_10      ;两个16位数分别右移2位
SHRU2   C_32,   A_22,   E_32
SHRU2   C_54,   A_22,   E_54
SHRU2   C_76,   A_22,   E_76
;;
PACKL4  E_32,   E_10,   D_3210    ;偶位字节提取组合成一个新数
PACKL4  E_76,   E_54,   D_7654
    ;;
STDWD_7654:D_3210,   *A_dst0++[A_pitch] ;存储8字节结果
```

```
    ;;[A_i]  BDEC    LOOP_HV_88, A_i
.return
.endproc
```

从以上线性汇编优化的过程分析可以看出，数据打包处理提高效率，编排使软件流水提高指令并行度。其他核心模块的汇编优化都可以根据这个指导思想来实现函数的高效运行。

根据 DM642 的处理能力，实现 4 路 CIF 或 1 路 D1 的 MPEG-4 SP 视频编码是完全可行的，前提是核心模块都作了汇编优化。

14.3 运行XviD CODEC系统

本书采用的 XviD CODEC 是 xvidcore-1.0.0 版本，XviD 网站有最新的代码 xvidcore-1.1.3。xvidcore-1.0.0 网址是 http://files.xvid.org/downloads/xvidcore-1.0.0.zip，作者使用迅雷下载软件成功下载。案例使用 VC++ 2005 和 CCS 2.2 软件运行 XviD CODEC。

14.3.1 YUV原始视频数据及其显示

MPEG-4 视频算法主要支持 I420 格式，即 YUV420 平面模式。网络上有许多此格式的标准测试序列文件：foreman.yuv、stefan.yuv、mobile.yuv 等，分辨率是 CIF（352 像素×288 像素），帧率是 25f/s。网站 http://trace.eas.asu.edu/yuv/index.html 有 QCIF、CIF 格式的 YUV 测试序列文件。

查看 YUV420/I420 的软件常用的是 YUVviewer 程序。光盘中有该程序的源代码，用户可以根据自己的需要进行修改。另外该程序也是 VC 6.0 平台下的，则在使用 VC 2005 运行时需要修改多个地方。

- yuvviewerdlg.cpp(641):error C2440: “static_cast” 无法从 “int (__thiscall CYUVviewerDlg::*)(void)”转换为“AFX_PMSG”。由于 VC++自动添加的消息处理函数返回值默认是 void，所以需要把函数 OnOpenfile()的返回值修改为 void。
- 警告“_CRT_SECURE_NO_DEPRECATE”：从前面的分析知道，这是从旧版本的 VC 程序移植到 VC++ 2005 时一个典型的警告，即 VC++ 2005 认为某些函数的使用不安全，因为程序对用户来说是清楚的，所以可以关闭该警告。增加宏定义，过程是在项目“属性→配置属性→C/C++→预处理器”中的预处理器定义中添加“;_CRT_SECURE_NO_DEPRECATE”。

14.3.2 VC平台下编译和运行XviD CODEC

由于 xvidcore-1.0.0 工程是 VC 6.0 平台，所以在使用 VC 2005 时，需要进行工程转换及修改各种格式错误。xvidcore 方案中有 4 个项目：libxvidcore 是 XviD CODEC，xvid_bench 是 CODEC 部分模块的检查程序，xvid_decraw 是 MPEG-4 视频解码，xvid_encraw 是 MPEG-4 视频编码。使用 VC++ 2005 运行 XviD 的过程如下。

STEP 01 启动 VC++ 2005 开发环境。

STEP 02 双击 xvidcore-1.0.0\build\win32\目录下的 xvidcore.dsw，则 VC++ 2005 提示如图 14-12 所示的画面：VC 6.0 项目转换为 VC 2005。

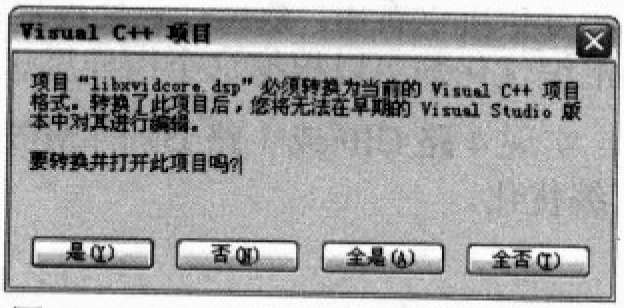


图 14-12　VC 6.0 项目转换为 VC 2005

STEP 03 单击图 14-12 中的“全是”按钮，把旧版的 VC 工程转换为 VC++ 2005 的项目。转换完成的 xvidcore 项目如图 14-13 所示。

STEP 04 选择图 14-13 中的“xvid_encraw”选项并单击鼠标右键，在弹出的快捷菜单中选择“设为启动项目”命令。

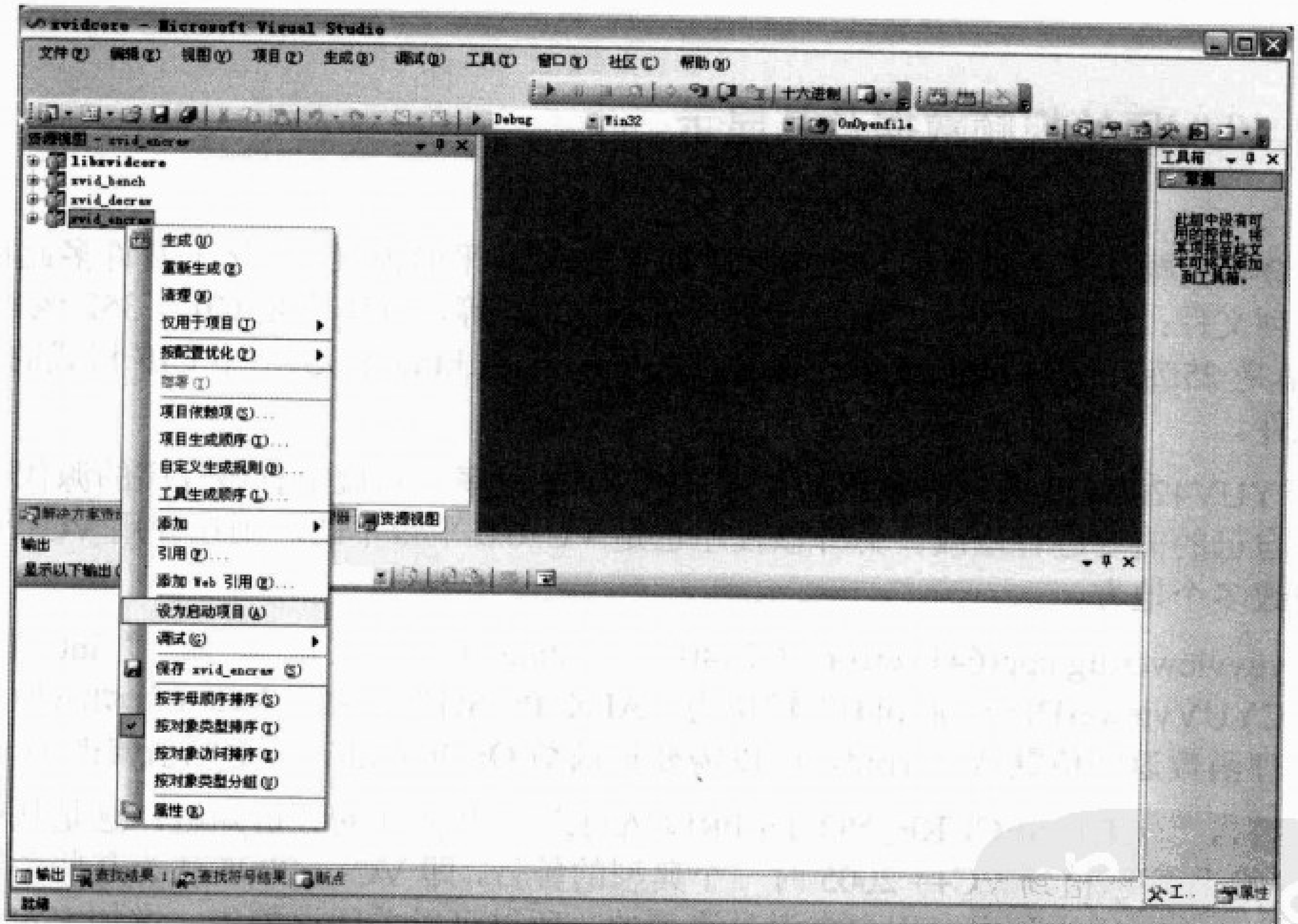


图 14-13　VC++ 2005 开发环境

STEP 05 生成 xvid_encraw。项目 xvid_encraw 依赖于 libxvidcore 库，所以要编译生成 xvid_encraw，则首先生成 libxvidcore。可能出现的问题及解决办法如下。

项目 libxvidcore 产生多个“_CRT_SECURE_NO_DEPRECATE”警告。我们可以通过配置“属性→C/C++→预处理器/预处理器定义”，来产生_CRT_SECURE_NO_ DEPRECATE。

如果 error C2632“int”后面的“int”非法，则可以通过文件 portab.h 屏蔽代码：

```
//#   define intptr_t int32_t
```

如果出现 nasm: error: no input file specified，则出错文件为\xvidcore-1.0.0\src\image\

x86_asm\colorspace_yuyv_mmx.asm。

这是因为旧版的 VC 在转换为 VC++ 2005 时，汇编文件的编译选项在转换过程中增加了双引号。我们可以通过更改出错文件 colorspace_yuyv_mmx.asm 的编译选项，选择该文件并单击鼠标右键，菜单“属性/自定义生成步骤/常规”，删除命令行中的双引号“””。

在清理解决方案时，会出现“vc80.idb 正在被使用”而无法删除的信息。即使关闭 VC 2005，该文件还是不能删除。这时候我们通过“工具→选项→项目和解决方案→生成并运行”→“最大并行项目执行数”命令将其修改为 1 即可。

STEP 06 为编码器控制台应用程序添加命令行参数。用鼠标右键单击 xvid_encraw 项目，按照路径“属性→配置属性→调试”，输入命令参数-asm -i e:\\yuv\\foreman.cif -o e:\\bitmp4\\foreman_enc.mp4 -type 0 -w 352 -h 288 -frames 300 -max_key_interval 100 -quality 2 -single -bitrate 300。

参数的意义见表 14-2 编码器参数说明。

表 14-2 编码器参数说明

名称	值	描述
-asm		是否使用平台 CPU 的汇编指令
-stats		输出重建图像 Y、U、V 的 PSNR 值
-i	e:\\yuv\\foreman.cif	输入的 YUV 视频文件
-o	e:\\bitmp4\\foreman_enc.mp4	输出的 MPEG-4 码流文件
-type	0	视频类型 0=I420
-w	352	视频图像宽度
-h	288	视频图像高度
-frames	300	要编码的视频帧数
-max_key_interval	100	I 帧最大间隔
-quality	2	编码图像质量，如 1/2 像素 MC、ADSP 的 ME
-single		使用码流控制
-bitrate	300	码流大小，单位为 Kbps

STEP 07 为解码器控制台应用程序添加命令行参数。用鼠标右键单击 xvid_decraw 项目，按照路径“属性→配置属性→调试”输入命令参数-asm -i e:\\bitmp4\\foreman_enc.mp4 -o e:\\bitmp4\\foreman_dec.yuv。参数的意义如表 14-3 所示。

表 14-3 解码器参数说明

名称	值	描述
-asm		是否使用平台 CPU 的汇编指令
-i	e:\\bitmp4\\foreman_enc.mp4	MPEG-4 码流文件
-o	e:\\bitmp4\\foreman_dec.yuv	解码后的 YUV 文件

14.3.3 CODEC在DSP下软仿真和硬仿真

算法在 CCS 开发软件下有两种运行环境：软件仿真和硬件仿真。算法研究或算法初期开发多数使用软件仿真，即 PC 模拟 DSP CPU。硬件仿真是程序在芯片中运行如 EVM 开发版，运行效果真实，硬仿也可以读写文件。

Xvidcore 1.0.0 是 VC 环境下的工程，为了能够使用 CCS 编译、优化该算法，需要建立 CCS 下的工程。在 13 章的案例中已经介绍了建立库和应用工程的过程，光盘是已经建立好的工程。

14.4 系统效果展示

XviD CODEC 实现的 MPEG-4 视频编/解码算法可以运行于各种 CPU 平台。在 CCS 平台实现编/解码算法时，应保证程序结果与 VC 平台下的相同。基于 DSP 平台优化视频算法时，一般都以 VC 平台结果为基准来调整 DSP 平台上的代码，优化算法提高其运行效率的同时应保证结果正确，这对于 DSP 平台的算法优化至关重要。

在 VC++ 2005 平台下实现 XviD CODEC 的视频编码，xvid_encraw 设为启动项目，XviD CODEC 视频编码如图 14-14 所示。

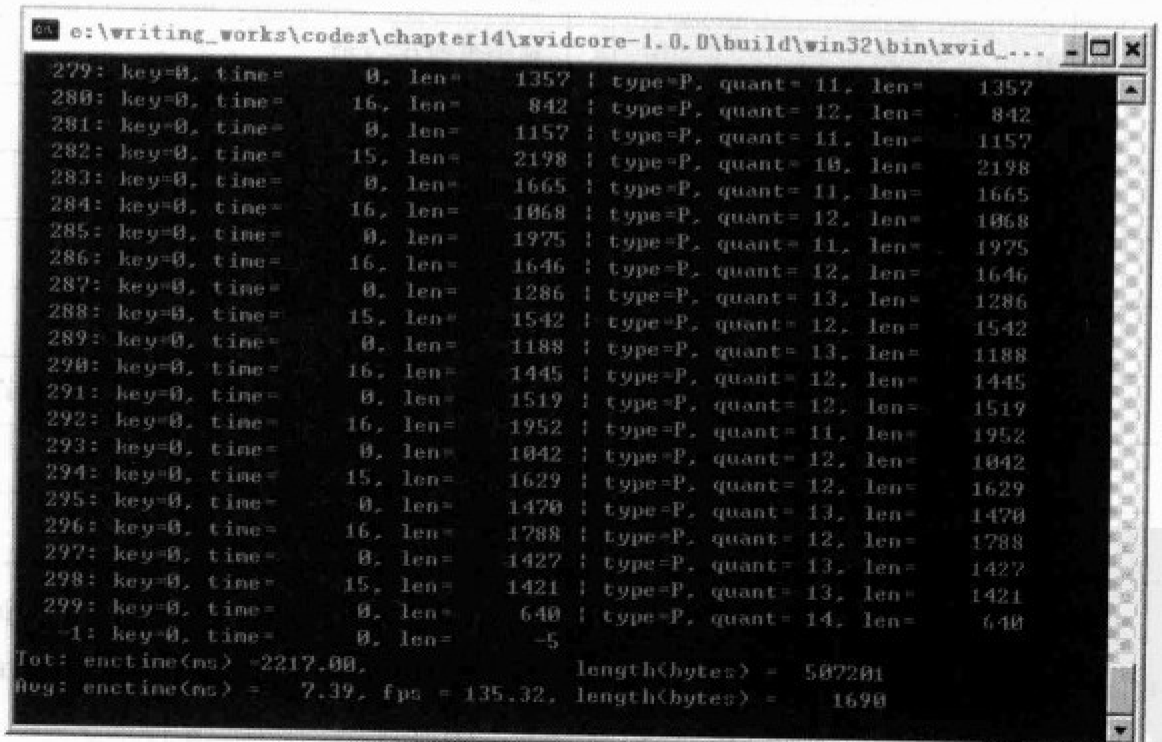


图 14-14　XviD CODEC 视频编码

输入参数：-asm -i e:\\yuv\\foreman.cif -o e:\\bitmp4\\foreman_enc.mp4 -type 0 -w 352-h 288 -frames 300 -max_key_interval 100 -quality 2 -single -bitrate 200000。参数意义如表 14-2 所示。

在 VC++ 2005 平台下实现 XviD CODEC 的视频解码，xvid_decraw 设为启动项目。XviD CODEC 视频解码如图 14-15 所示。

输入参数：-asm -i e:\\bitmp4\\foreman_enc.mp4 -o e:\\bitmp4\\foreman_dec.yuv。参数意义如表 14-3 所示。

原始 YUV 图像和经过 MPEG-4 编/解码后的图像如图 14-16 所示，图（b）为编/解码后的图像，与图（a）原始图像相比质量略有下降。

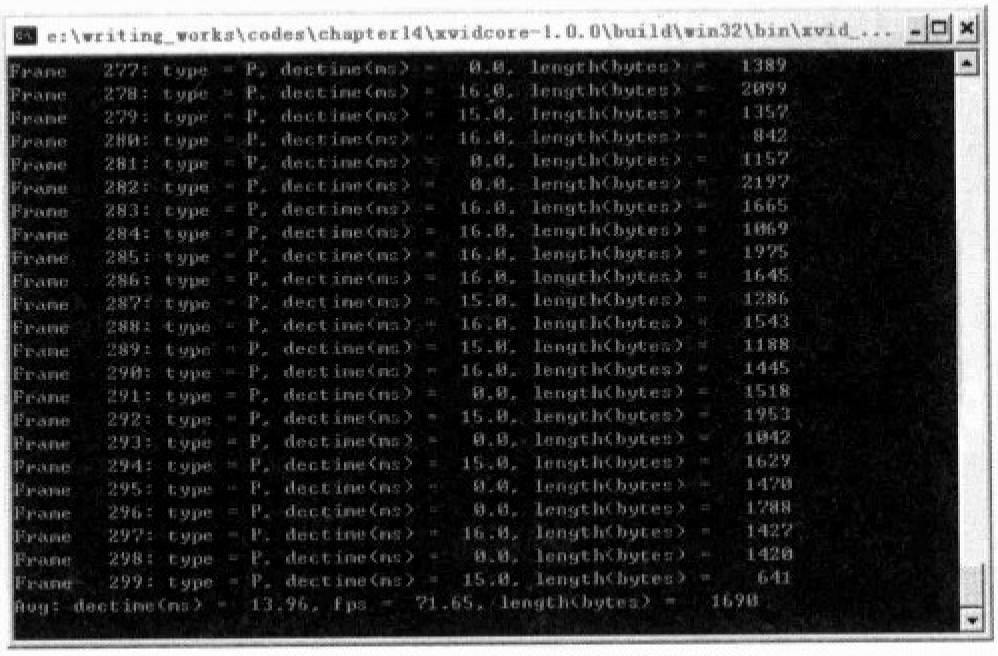


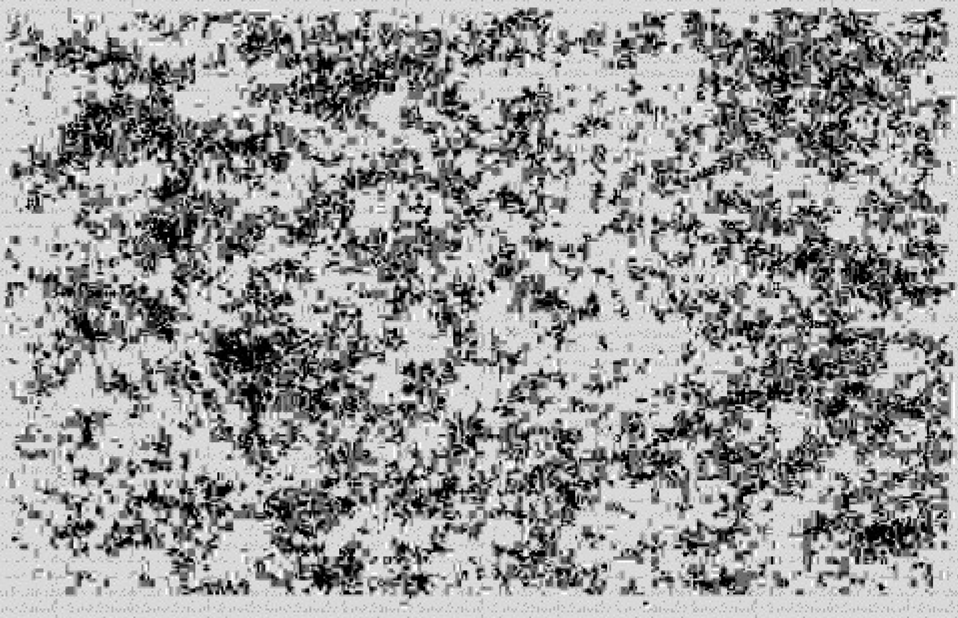
图 14-15 XviD CODEC 视频解码

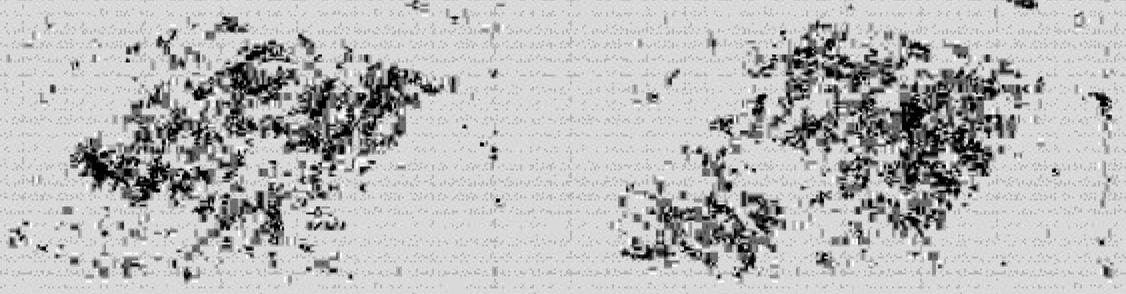
（a）原始 YUV 图像　　（b）编/解码后图像

图 14-15 MPEG-4 编/解码前后图像对比

14.5 本章小结

本章首先介绍了 MPEG-4 视频编码算法的原理，对 MPEG-4 编码算法的工程实现 XviD CODEC 的工作过程做了详细的介绍。首先 XviD 编码算法层次清晰，编码器以 3 个不同的参数一个函数调用的形式完成编码器的使用。MPEG-4 SP 编码分 I 帧和 P 帧编码，I 帧以 Intra 模式、宏块为单位进行编码。XviD 解码算法组织形式同编码算法。详细介绍了编/解码算法底层函数的 MMX/SSE 媒体汇编和 DM642 的线性汇编优化方法。最后介绍了 XviD CODEC 在 VC++2005 平台下编译、运行的过程，对常见错误做了分析和解答，同时展示了编/解码前后的图片效果。





第 15 章

X264 实现 H.264/AVC 视频编码

随着 NGN（下一代网络）、3G 及 3G 演进和 NGBW（下一代宽带无线）等对视频、多媒体业务与网络应用的需求飞速发展，作为视频业务及存储应用核心技术的高效率视频数字压缩编/解码技术，已成为目前广播、视频与多媒体通信领域中的亮点与热点。

制定视频编码标准的两大国际组织为 ITU-T 的视频编码专家组（Video Coding Expert Group，VCEG）及 ISO/IEC 的活动图像专家组（Moving Picture Expert Group，MPEG）。这些专家组针对不同的视频应用领域制定了视频压缩编/解码标准，同时他们还联合制定了高级视频编/解码标准 H.264/AVC。本章就通过实际的应用案例去讲解 H.264/AVC 视频编码实现技术。

本章重点

- H.264/AVC 编码概要
- X264 视频编码分析
- 运行 X264 编码系统
- 系统效果展示

15.1 H.264/AVC编码概要

H.264/AVC 协议与 MPEG-4 视频编码标准类似，同样只规定了码流的语法和意义，并提供了软件校验模型 JM，没有规定具体的、实用的工程实现。开源的 H.264/AVC 编/解码算法工程实现有 JM、X264 及 T264，解码工程实现有 JM、ffmpeg 及 T264。JM 参考软件可以用来验证码流，但其工程复杂编/解码效率非常低，不过一直有更新。T264 是国内的开源 H.264/AVC 编/解码，其码流不完全符合标准，图像效果相对较差，2005 年停止更新。X264 是法国的开源 H.264/AVC 编码，码流标准、图像效果好，国内的研究者几乎都以此为蓝本来设计和优化符合特定应用的 H.264/AVC 编/解码算法，该开源编码算法最近一直有更新。

另外，X264 的命名有些类似于 MPEG-4 视频算法的工程实现 XviD，“X”意味着所有。并且 X264 的工程框架和 XviD 非常相似。T264 更是 XviD 的“翻版”，两者不少的底层函数相同，可见 XviD 的“魅力”所在。

15.1.1 开源代码工程管理软件

几乎所有的开源算法工程都是使用专门的下载、版本管理软件来实现的，如 CVS、SVN（Subversion）、Git 等。以前的 X264 使用 CVS、SVN 等下载软件下载，现在最新的 X264 开源代码需要 Git 来下载。Git——The Stupid Content Tracker，傻瓜内容跟踪器。Git 是用于 Linux 内核开发的版本控制工具，与常用的版本控制工具 CVS、SVN 等不同，它采用了分布式版本库的方式，不需要服务器端的软件支持，使源代码的发布和交流极其方便。Git 的下载速度很快，这对于诸如 Linux 这样的大项目来说自然很重要。

15.1.2 获取开源算法工程

JM 是 ITU-T 提供的 H.264/AVC 参考软件，运行效率极低。可以作为 H.264/AVC 码流的验证模型，它包括视频编码和解码。JM 的下载地址为 http://iphome.hhi.de/suehring/tml/。JM 包含编码 lencode 和解码 ldecode 两个工程，使用 VC++平台编译。

X264 的下载需要专门的下载软件，如 CVS、SVN、Git，这些本来是 Linux 环境下的应用软件，不过 Windows 下也有对应的版本。Git 的下载地址为 http://code.google.com/p/msysgit/downloads/list，选择下载应用程序的最新版本。X264 的相关网站为 http://www.videolan.org/developers/x264.html。使用 Git 获取 X264 的命令：#git clone git://git.videolan.org/x264.git。X264 提供了 H.264 的视频编码算法，没有提供解码。X264 工程使用 VC++平台编译。

网站 http://sourceforge.net 集合了几乎所有的开源工程代码，T264 可以从该网站上下载：http://sourceforge.net/projects/t264/。不过自 2005 年以来已没有更新了。T264 工程使用 VC++、TI 的 CCS 平台编译。

音/视频解码 CODEC 开源集合 ffmpeg，集成了多种媒体格式的解码回放源代码。ffmpeg 中的 H.264 解码部分以易于硬件、软件编程实现而被大量应用。下载 ffmpeg 开源集合可以使

用 SVN 或 Git。使用 SVN 下载 ffmpeg 最新源代码为 svn://svn.mplayerhq.hu/ffmpeg/trunk；使用 Git 下载 ffmpeg 最新源代码为#git clone git://git.mplayerhq.hu/ffmpeg/。ffmpeg 工程使用 Linux 环境编译和调试，尽管有 Windows 下的工程，但是不能使用 VC++平台编译和调试。

15.1.3 H.264/AVC新特性

H.264/AVC 是新一代视频编/解码标准，提供了诸多以往视频算法所没有的新特性，以提高编码效率，这些新特性如下。

- 利用临近块的边界像素的 Intra 空间预测，提供 16×16、8×8、4×4 预测模式。
- 多参考帧的运动补偿。比起以前的视频编码标准，H.264/AVC 以更灵活的方式使用更多的重建帧作为参考帧。在某些情况下，最多可以使用 32 个参考帧（在以往标准中，P 帧的参考帧数目为 1，B 帧的参考帧数目为 2）。该特性对大多数场景序列都可以带来一定的码率降低或者质量提高，对某些类型的场景序列，例如快速重复的闪光，反复的剪切或者背景遮挡的情况，它能很显著地降低编码的码率。
- 变块尺寸运动补偿。可使用最小 4×4 至最大 16×16 的块来进行运动估计与运动补偿，能够对图像序列中的运动区域进行更精确的分割。
- 完全可逆的整数 DCT4×4 变换，同样在高精度拓展中，采用整数 8×8 变换，并能在 4×4 变换和 8×8 变换中进行自适应的选择。
- 在第一次 4×4 变换后，再对 DC 系数（亮度、色度的 DC 系数）进行一次 Hadamard 变换，使得在平滑区域得到更好的压缩效果。
- 为了减少锯齿（Aliasing）并得到更锐化的图像，采用六抽头的滤波器来产生 1/2 像素的亮度分量预测值。
- 宏块对结构允许场模式中采用 16×16 的宏块（相对于 MPEG-2 中的 16×8）。
- 1/4 像素精度的运动补偿能够提供更高精度的运动块预测，由于色度通常是亮度抽样的 1/2（YUV420/I420），因此运动补偿的精度就达到了 1/8 像素精度。
- 加权的运动预测，指在运动补偿时可以使用增加权重和偏移的办法。它能在一些特殊的场合，如淡入、淡出、淡出而后淡入等情况下提供很大的编码增益。
- 使用了一个环内的去除块效应滤波器，能够减轻普遍存在于其他基于离散余弦变换（DCT）的视频编/解码器的块效应。
- 基于上下文的变长编码（CAVLC），用于对量化后的变化系数进行编码。比起 CABAC，它的复杂度相对较低，压缩比不高，但是比起以前的视频编码标准所使用的熵编码方案，它又是很有效的。
- 基于上下文的二元算数编码（CABAC），它能够灵活地将各种语法元素在已知相应上下文概率分布的情况下进行更有效的无损熵编码。
- 对既不是用 CABAC 也不是用 CAVLC 的语法元素使用 Exponential-Golomb（Exp-Golomb）熵编码方案进行编码。
- 使用一个网络抽象层（NAL），使得相同的视频语法可以适用于多种网络环境中；并且使用了序列参数集（SPSs）和图像参数集（PPSs）来提供更高的鲁棒性

（Robustness）和灵活性。

- 帧切换/切换片（Switching Sices，包括 SP、SI），它使得编码器能够指令解码器跳转到一个正在处理的视频码流，用来解决视频码流码率切换。当解码器利用 SP/SI 条带跳转到一个视频码流中间时，除非之后的解码帧引用切换帧之前的图像作为参考帧，它都可以得到完全一致的解码重建图像。
- 灵活的宏块排列模式 FMO（Flexible Macroblock Ordering，也称为片组 Slice Groups 技术）和任意条带排列 ASO（Arbitrary Slice Ordering）模式，用来更改图像编码的最基本单位——宏块的编码顺序。它能够用来提高有扰信道下码流的鲁棒性（Robustness）及一些其他的目的。
- 数据分区 DP（Data Partitioning）能够将不同重要程度的语法元素分开打包传输，并使用非平等数据保护 UEP（Unequal Error Protection）等技术来改善视频码流对抗信道误码/丢包的鲁棒性（Robustness）。
- 冗余片 RS（Redundant Slices）同样是一个提高码流鲁棒性的技术。编码器利用该技术可以发送图像某区域（或者全部）的另一个编码表示（通常是较低分辨率的编码码流），使得当主要信息发生错误或者丢失的时候能够用冗余的第 2 个编码表示来解码。
- 使用了一个自动的字节码流打包方法，避免了码流中出现与开始码重复的码字。开始码是码流中用于随机访问和重建同步的码字。
- 补充增强信息 SEI（Supplemental Enhancement Information）和视频可用信息 VUI（Video Usability Information）增加了向视频码流中加入信息的办法，为各种应用提供了途径。
- 辅助图层（Auxiliary Pictures）可以用来实现某些特殊的功能，例如 alpha 复合（alpha compositing）。
- 帧编号，使用该功能支持创建一个视频序列的子序列，可用来支持实现时域的可伸缩性，还支持对丢失的整帧图像进行检测和隐藏（丢失可能是由网络丢包或者信道误码造成的）。

15.2 X264视频编码分析

H.264/AVC 视频协议类似于 MPEG-4 协议，也分为多个档级标准以适应不同的领域应用。H.264/AVC 定义了 4 种不同的 Profile（类）：Baseline Profile（基类）、Main Profile（主类）、Extended Profile（扩展类）和 High Profile（高端类）。

- Baseline Profile 提供 I/P 帧，仅支持 Progressive（逐行扫描）和 CAVLC。主要应用在“视频会话”中，如可视电话、会议电视、远程医疗、远程教学、视频监控等，要求低延迟。
- Extended Profile 提供 I/P/B/SP/SI 帧，仅支持 Progressive（逐行扫描）和 CAVLC。主要应用于网络的视频流，如视频点播、基于网络的视频监控等。
- Main Profile 提供 I/P/B 帧，支持 Progressive（逐行扫描）和 Interlaced（隔行扫描），

提供 CAVLC 或 CABAC。主要应用于消费电子应用，如数字电视广播、数字视频存储等。

- High Profile（即 FRExt），在 Main Profile 基础上新增 8×8 帧内预测（8×8 Intra Prediction），自定义量化（Custom Quant），无损视频编码（Lossless Video Coding），更多的 YUV 格式（4:2:2、4:4:4），像素精度提高到 10 位或 12 位等。主要应用于对高分辨率和高清晰度有特别要求的专业应用。

X264 是 H.264/AVC 视频压缩标准的一个工程实现，它最大的特点是为了 H.264/AVC 的实用性做了部分功能简化。与 JM 相比，X264 并没有完整地实现 H.264/AVC 标准，而是舍弃了一些对编码性能贡献很小但计算复杂度极高的新特性，如多参考帧、帧间预测中不必要的块模式等技术。从而极大地降低了计算的复杂度，编码效率却并没有明显降低。X264 主要实现了 Main 档级的功能，码流兼容 H.264/AVC。在实用的系统中，比如比较普遍的在 DSP 芯片上实现 H.264/AVC，大多是移植 X264 工程，且实现 Baseline 或 Main 档级。

15.2.1 H.264/AVC关键技术要点

H.264/AVC 视频编码标准的数据处理流程仍然是混合编码框架，即基于像素块的运动补偿、变换、量化和熵编码的处理模块。H.264/AVC 算法的这些处理模块都做了技术改进，以提高编码效率。典型的视频编码框架如图 15-1 所示。

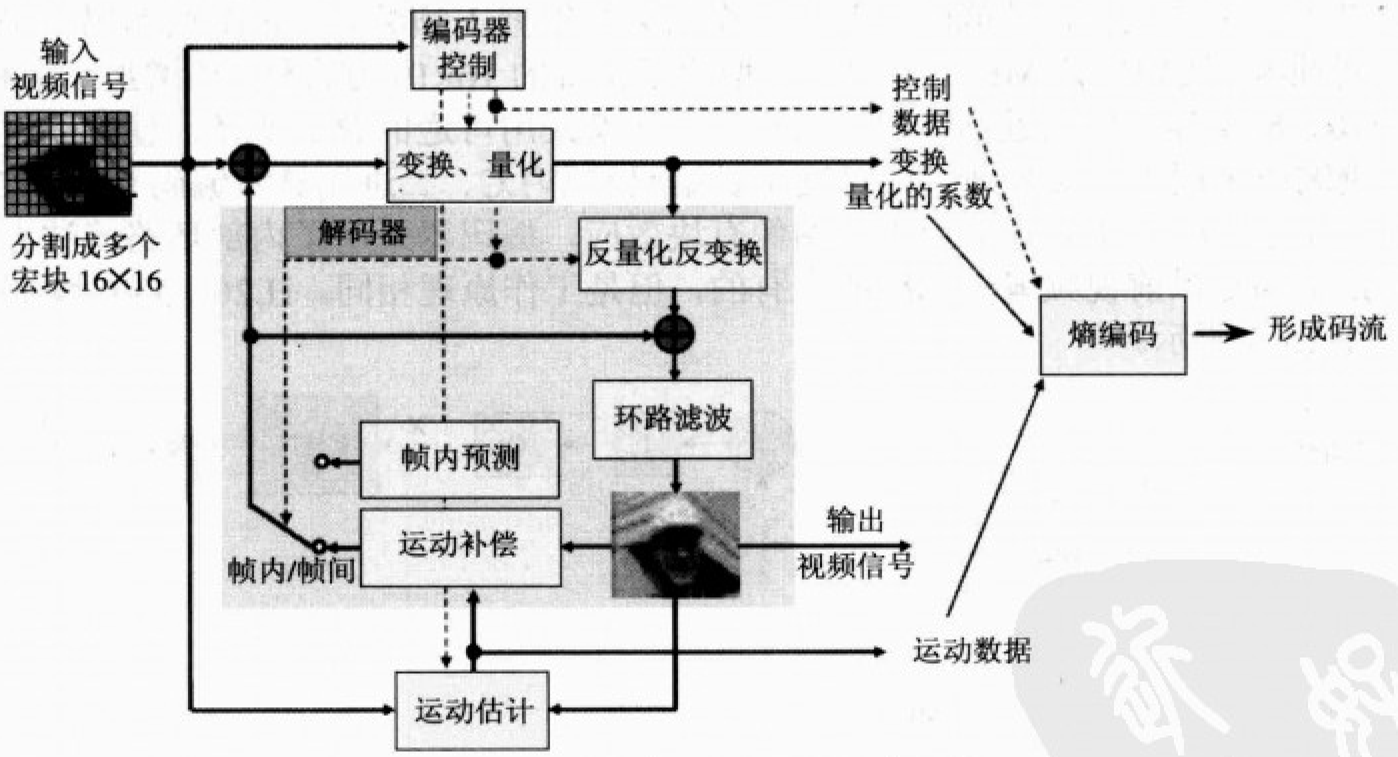


图 15-1 混合视频编/解码技术框架

处理的图像帧分割成多个 16×16 的宏块，根据图像编码类型 I 帧、P 帧或 B 帧等，决定是对图像数据本身处理还是对残差编码。图像数据本身即帧内编码 I 帧，利用图像的空间相关性去除空间冗余。残差编码首先利用运动估计技术，在一定范围的窗口内搜索相对最佳宏块，然后相减形成残差，同时记录当前宏块的运行向量 MV。对图像数据或残差做 DCT 变换、量化。为了解码器解码时图像不产生偏移，还要做 IDCT 变换、反量化，重建形成参考帧。由于块处理造成“块效应”，因此采用环路滤波来提高图像的清晰度。最后对量化后的系数、运动向量及其他信息做熵编码，形成最后的码流。

H.264/AVC 将视频压缩系统分成了视频编码层 VCL（Video Coding Layer）和网络提取层 NAL（Network Abstract Layer），其结构如图 15-2 所示。视频编码层 VCL 通过时域、空域预测和变换编码来完成对视频信息的压缩。而网络提取层 NAL 则将与网络相关的信息从视频压缩系统中抽象出来，使网络对于视频编码层是透明的，它主要负责对编码后的数据进行封装以提高其鲁棒性、降低误码率。采用分层结构有利于压缩编码与网络传输之间的分离，使视频编码层可以移植到不异构的网络中，适应不同的网络传输。

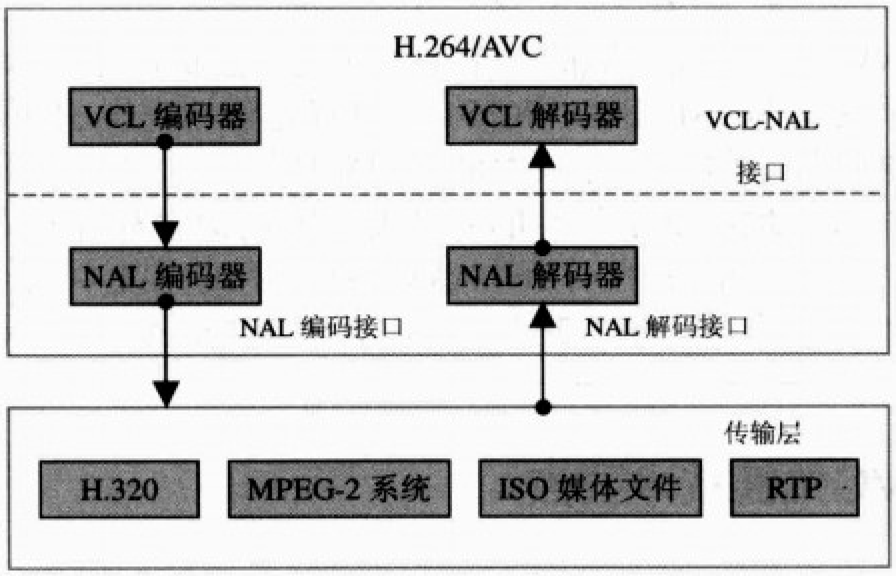


图 15-2 H.264 传输结构

视频编码层 VCL 是 H.264/AVC 的核心技术，视频算法是混合编码技术框架，变换、量化、编码 3 个步骤。不过每个过程有了新的技术，码流更小、图像效果更好。每个模块的小的技术改进带来了系统整体性能的提升。变换的数据根据 I 帧编码、P/B 帧编码，采用帧内预测和帧间预测（运动估计 ME）。宏块的类型除了传统的 16×16 方式外，还增加了多种类型：16×8、8×16、8×8 等，更贴近物体的运动模型。变换采用可逆的整数变换，使解码过程无误差积累。熵编码采用上下文自适应的可变字长编码 CAVLC、二进制算术编码 CABAC，码流更低。另外由于 DCT 块处理的原因，图像有块效应，编码器采用了内部环路滤波去除块效应。这些技术都是以前视频编码标准所没有的，但是工作原理相同。H.264/AVC 编码器的工作框图如图 15-3 所示。

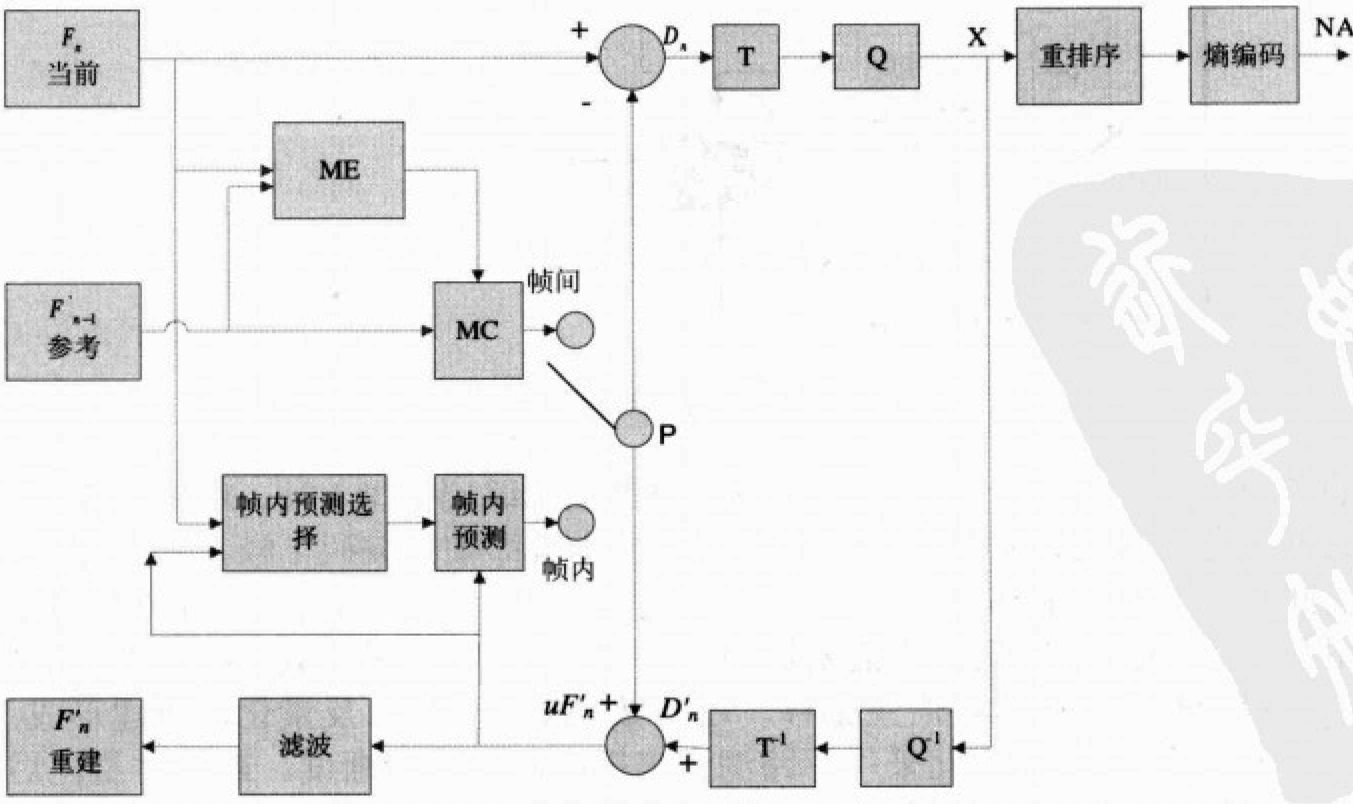


图 15-3 H.264/AVC 编码器功能框图

图 15-3 中输入的帧 F_n 以宏块为单位被编码器处理。首先，按帧内或帧间预测的编码方法进行处理。如果用帧内预测编码，则其预测值 P 是由当前片中已编码的图像直接预测得到的。如果采用帧间预测编码，则其预测值 P 是由当前片中已编码的参考图像经运动补偿（MC）后得到的，其中参考图像用 F'_{n-1} 表示。为了提高预测精度，增加压缩比，实际的参考图像可在过去或未来的已解码重建、滤波的帧中进行选择。

预测值 P 和当前块相减后，产生一个残差块 D_n，经块变换、量化后产生一组量化后的系数 X，再经熵编码与解码所需的一些边信息一起组成一个压缩后的比特流，经 NAL 层封装后供传输和存储使用。为了提供预测用的参考图像，编码器必须有重建图像的功能。因此必须使残差图像经反量化、反变换后得到的 D'_n 与预测值 P 相加，得到 uF'_n。为了去除编码解码环路中产生的噪声，提高参考帧的图像质量，从而提高压缩图像性能，设置了环路滤波器，滤波后的输出 F'_n 即为重建图像，可用做参考图像。

1. 帧内预测

对 I 帧的编码是通过利用空间相关性来实现的。以前的视频标准只利用了一个宏块 MB 内部的相关性，而忽视了宏块之间的相关性，所以编码后的数据量较大。为了能进一步利用图像空间的相关性，H.264/AVC 引入了帧内预测以提高压缩效率。总体来说，帧内预测编码就是用周围邻近的像素值来预测当前的像素值，然后对预测误差进行编码。帧内预测是基于图像块的，亮度分量（luma）和块的大小可以在 16×16 和 4×4 之间选择，16×16 块有 4 种预测模式（垂直 vertical、水平 horizontal、直流分量 DC、平面 plane），4×4 块有 9 种预测模式（直流分量 DC、8 个方向）。对于色度分量（chroma），预测是对整个 8×8 块进行的，有 4 种预测模式（垂直 vertical、水平 horizontal、直流分量 DC、平面 plane）。除了 DC 预测外，其他每种预测模式对应不同方向上的预测。

1）16×16 亮度预测模式

16×16 亮度预测信号根据预测的方向不同，有 4 种预测模式，其预测模式如图 15-4 所示，分别是垂直方向、水平方向、DC、平面模式。

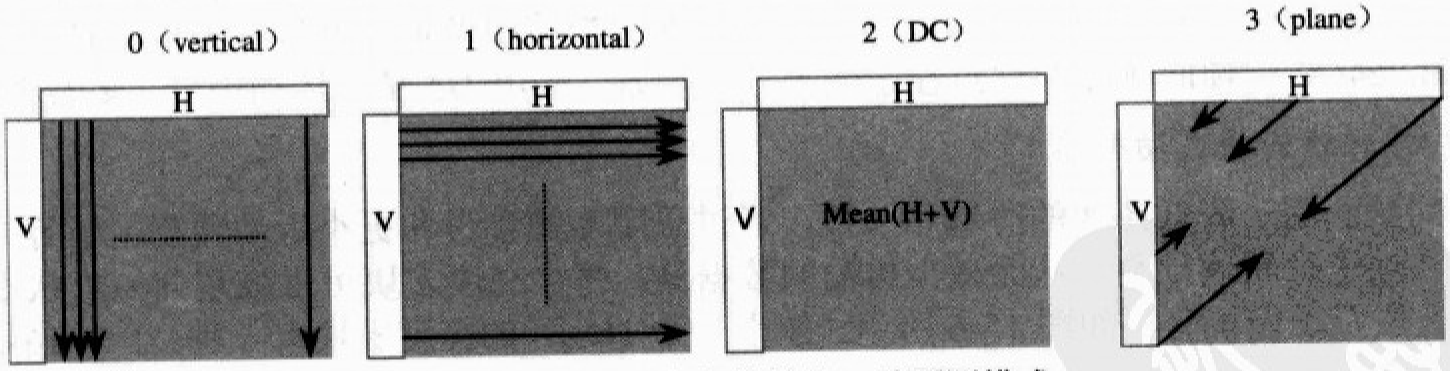


图 15-4　16×16 亮度信号的 4 种预测模式

2）4×4 亮度预测模式

在 4×4 亮度预测模式中，H.264/AVC 把 16×16 的宏块划分成 16 个 4×4 的子块，每个子块用与其相邻的上面和左面已经编码并重建后的像素点进行预测，如图 15-5 所示。

4×4 亮度信号根据预测方向的不同共有 9 种不同的预测模式。其预测模式如图 15-5 所示。其中，垂直预测（模式 0）、水平预测（模式 1）、DC 预测（模式 2）总是被认为是有效的，即使是在上边或左边像素值不存在的情况下，可认为不存在的像素值为 128 进行填充，来进行预测。而其他模式则只有在其需要使用的像素值存在的情况下才能进行预测。

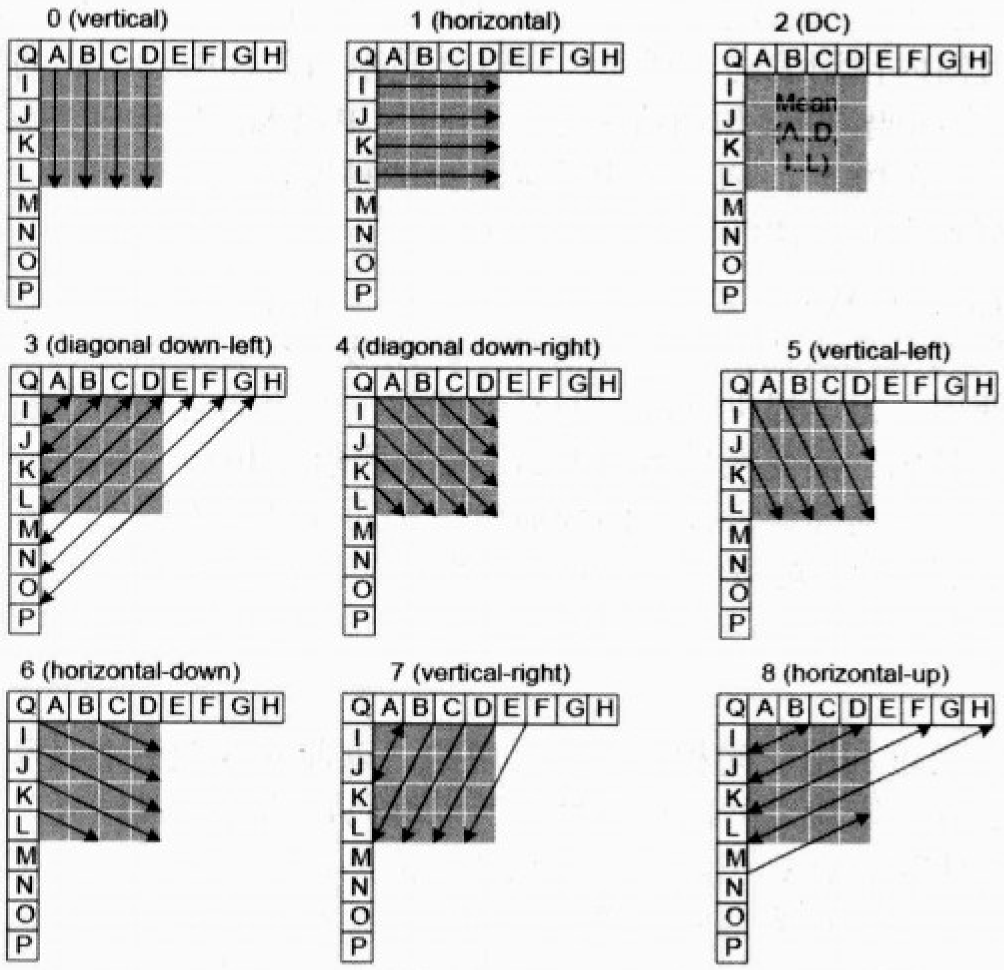


图 15-5　4×4 亮度信号的 9 种预测模式

3）8×8 色度预测模式

色度分量以 8×8 块为单位进行预测，它与 16×16 亮度预测模式类似。其中相同位置的两个色度块 U/C_b、V/C_r 具有相同的最佳预测模式。

2．帧间预测

帧间预测是利用先前编码帧的重建图像作为参考，对当前图像进行预测编码的一种方式。它把参考图像的抽样点通过运动矢量的补偿作为当前图像抽样值的参考值。和以往标准相比，H.264/AVC 帧间预测的优点在于块尺寸更丰富，采用 1/4 精度运动矢量、多参考帧等。

1）树状结构的运动补偿

帧间预测用于降低图像的时域相关性，通过采用多帧参考和更小运动预测区域等方法，来对下一帧进行精确预测，从而减少传输的数据量。每个亮度宏块可以被划分为形状不等的区域。宏块及子宏块分割如图 15-6 所示。图中一个 16×16 的亮度宏块有 4 种：1 个 16×16、2 个 16×8、2 个 8×16、4 个 8×8。而 8×8 模式的每个子宏块可以进一步进行分割，也有 4 种：1 个 8×8、2 个 4×8、2 个 8×4 及 4 个 4×4。

每个分割或子宏块都有一个独立的运动补偿。每个运动矢量都须被编码、传输，分割的选择也要编码压缩到比特流中。对于较大的分割尺寸，运动矢量的选择和分割的选择只需少量的比特，但运动补偿残差较多。小尺寸的分割可以减少运动补偿后的残差，但是需要传输较多的运动矢量和分割的选择。因而，大尺寸分割适合于较平坦的区域，小尺寸分割适合于细节较多的区域。

宏块的色度成分（C_r,C_b）为相应亮度的一半，采用和亮度相同的分割模式，只是尺寸减半。

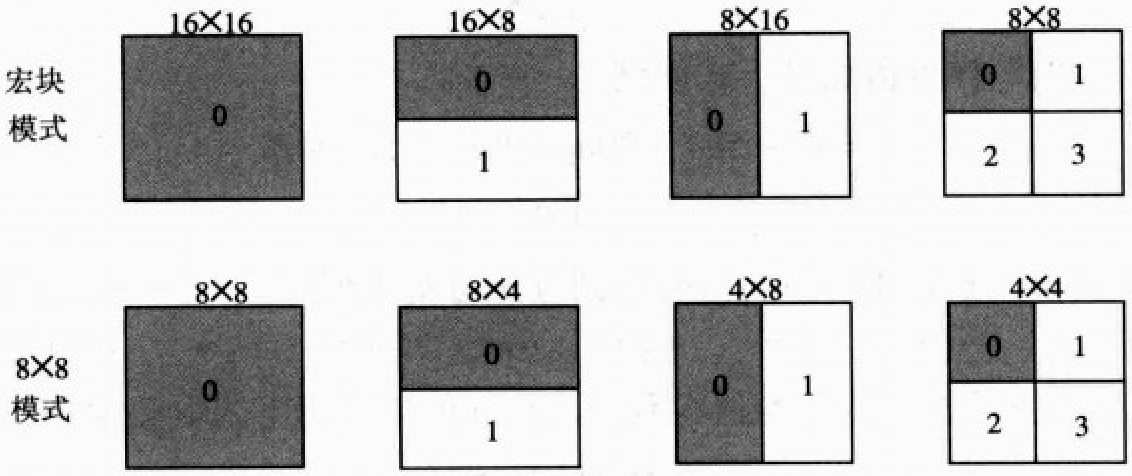


图 15-6　宏块及子宏块分割

2）1/4 精度运动矢量

H.264/AVC 中的运动矢量对亮度信号采用 1/4 精度。当运动矢量指向整数抽样点位置时，预测值将直接从相应的参考图像抽样获取。当运动矢量指向分数抽样点时，其分数位置的像素并不存在于参考图像中，需利用邻近已编码点进行内插得到。内插像素的生成方法如图 15-7 所示。

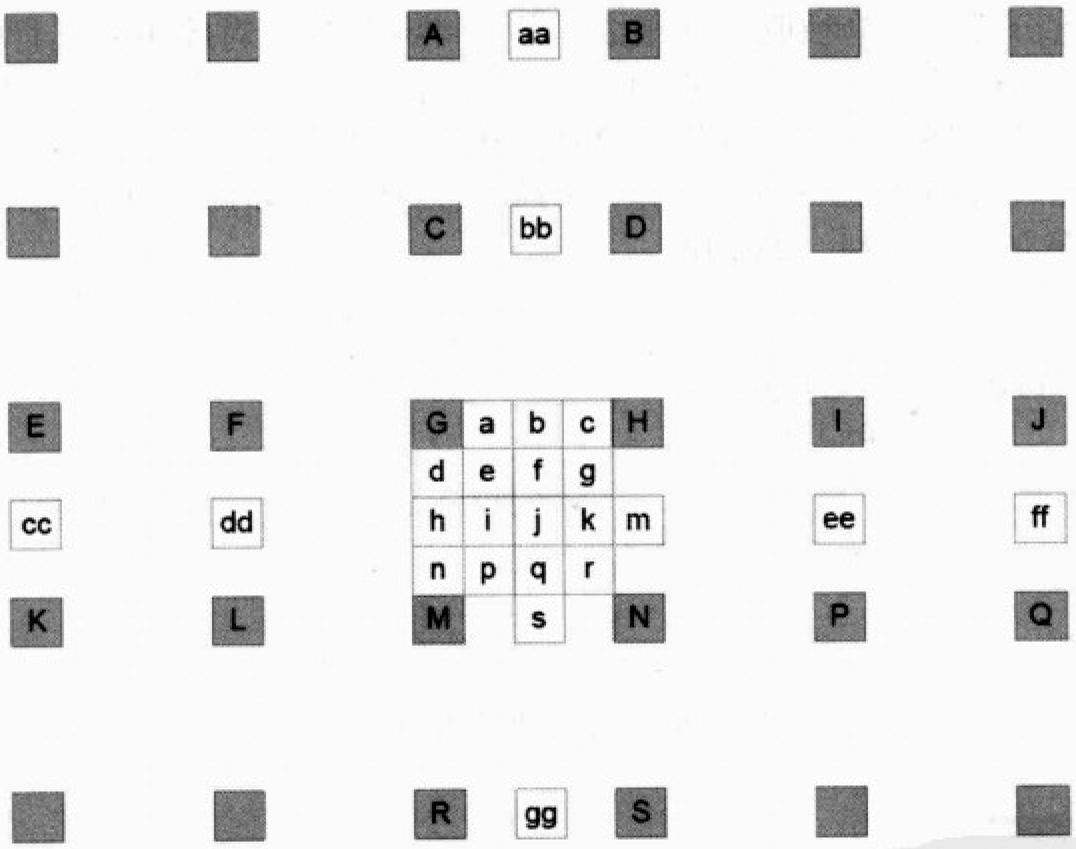


图 15-7　内插像素的生成方法

在分数像素位置内插中，1/2 位置的采样信号的预测值是通过应用一个一维的水平和垂直六阶 FIR 滤波器得到的。1/4 像素位置的采样信号的预测值是通过将整数和 1/2 像素采样位置的采样点插值得到的。

在图 15-7 中，1/2 位置的“b”、“h”，先运用一个 6 抽头的滤波器（1、−5、20、20、−5、1）计算得到中间值 b_1、h_1 如下所示：

$$b_1 = (E - 5F + 20G + 20H - 5I + J) \tag{15-1}$$

$$h_1 = (A - 5C + 20G + 20M - 5R + T) \tag{15-2}$$

对滤波后得到的值进行归一化处理，然后取整，最后把值限制在[0,255]之间，所以：

$$b=(b_1+16)>>5 \tag{15-3}$$

$$h=(h_1+16)>>5 \tag{15-4}$$

在 1/2 位置的"j"样点值通过下式得到：

$$j_1 = cc - 5dd + 20h_1 + 20m_1 - 5ee + ff \tag{15-5}$$

$$j = (j_1 + 512) >> 10 \tag{15-6}$$

其中 cc、dd、ee、m_1、ff 的中间值的得到方法与 h_1 类似。

1/4 位置的"a"、"c"、"d"、"n"、"f"、"i"、"k"、"q"样点值是通过对邻近的整数或 1/2 位置的样点取平均值得到的。如 a 的取值用下式取整得到：

$$a=(G+b+1)>>1 \tag{15-7}$$

相应的色度像素采用 1/8 精度的运动矢量，也是通过整数像素的线性内插得到的。

3. 整数变换 DCT 及量化

H.264/AVC 中的宏块大小为 16×16，对其中每个 4×4 大小的块进行 4×4 的整数 DCT 变换后，得到 16 个 4×4 的变换矩阵。为了进一步提高压缩效率，该协议还允许把每个 4×4 的变换矩阵中的直流分量 DC 单独取出并组成一新的 4×4 矩阵，对此矩阵进行 Hadamard 变换。整数 DCT 避免了以往标准中的编/解码的不匹配问题，使得反变换不会出现失衡的问题。并且其运算只包含加减和移位，且将量化融合在其中，有效地降低了运算量。H.264/AVC 编码中整数变换及量化过程如图 15-8 所示。宏块的数据传送顺序如图 15-9 所示，图中的上方−1、16、17 为直流分量形成的 4×4、2×2 块。

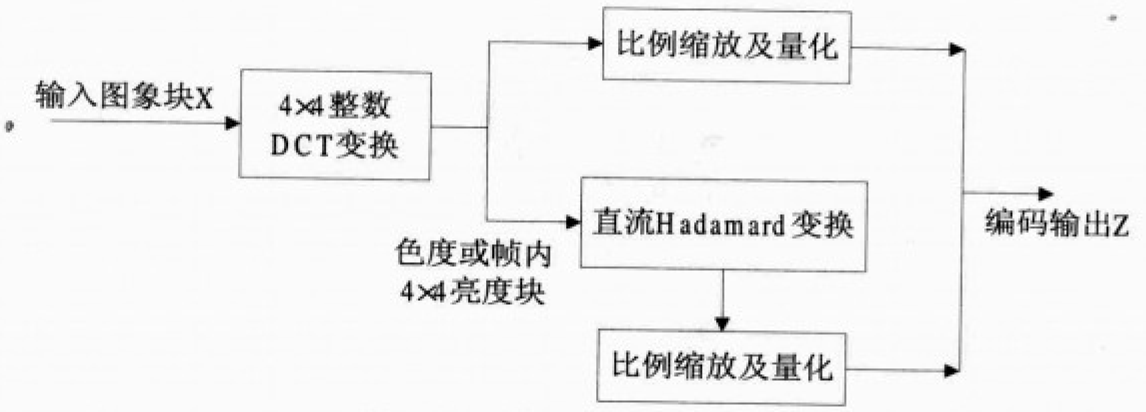


图 15-8　整数变换及量化过程

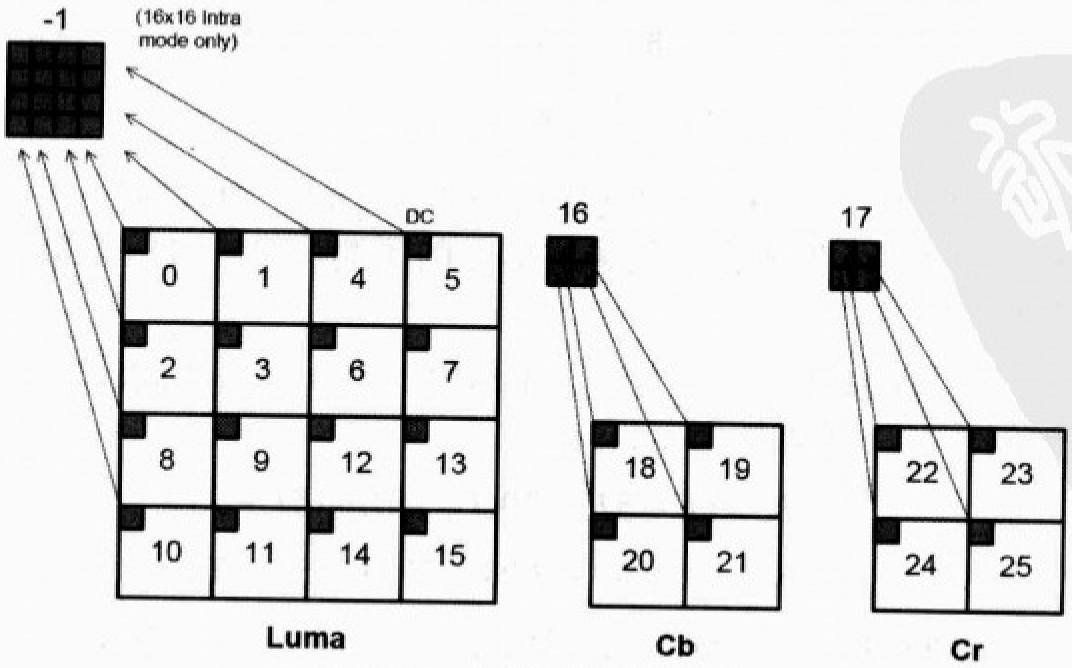


图 15-9　宏块数据传送顺序

哈达码变换 Hadamard 是用简单的变换估算变换后产生的比特数。在 H.264/AVC 的运动估计中用哈达码变换替代 SAD，如果要求设计低代价硬件，则可以将这部分省略。

公式 15-8 为 DCT 变换过程，其中 X 为待变换的数据，$\otimes \mathrm{E_f}$ 运算的乘法融合到后面的量化过程中，在整数 DCT 变换中不需要计算，实际计算的只是 $W = C_f X C_f^T$。

$$Y = (C_f X C_f^T) \otimes E_f = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} X \begin{bmatrix} 1 & 2 & 1 & 1 \\ 1 & 1 & -1 & -2 \\ 1 & -1 & -1 & 2 \\ 1 & -2 & 1 & -1 \end{bmatrix} \right) \otimes \begin{bmatrix} a^2 & \frac{ab}{2} & a^2 & \frac{ab}{2} \\ \frac{ab}{2} & \frac{b^2}{4} & \frac{ab}{2} & \frac{b^2}{4} \\ a^2 & \frac{ab}{2} & a^2 & \frac{ab}{2} \\ \frac{ab}{2} & \frac{b^2}{4} & \frac{ab}{2} & \frac{b^2}{4} \end{bmatrix} \tag{15-8}$$

H.264/AVC 采用标量量化，其量化过程表示为：

$$Z_{ij} = round\left(\frac{Y_{ij}}{Qstep}\right) \tag{15-9}$$

其中，Y_{ij} 是上述矩阵 Y 中的转换系数，Z_{ij} 是输出的量化系数，$Qstep$ 是量化步长。在 H.264/AVC 中量化步长共有 52 个值，QP 为量化参数，它是量化步长的序号（0～51），QP 越小表示量化越精细，QP 越大表示量化越粗糙。在量化表中，QP 每增加 6，$Qstep$ 就增加一倍。

4．熵编码

H.264/AVC 协议对于图像数据或残差提供了两种熵编码的方式：基于上下文自适应变长码 CAVLC（Context-based Adaptive Variable Length Coding）和基于上下文自适应二进制算术编码 CABAC（Context-based Adaptive Binary Arithmetic Coding）；如果不是预测残差而是运动向量等其他数据，则 H.264/AVC 采用 Exp-Golomb 码或 CABAC 编码，视编码器的设置而定。

1）CAVLC

变长编码 VLC 的基本思想就是对出现频率大的符号使用较短的码字，而出现频率小的符号采用较长的码字，这样可以使得平均码长最小。在 CAVLC 中，H.264/AVC 采用若干 VLC 码表，不同的码表对应不同的概率模型。编码器能够根据上下文，如周围块的非零系数或系数的绝对值大小，在这些码表中自动地选择，最大可能地与当前数据的概率模型匹配，从而实现了上下文自适应的功能。

2）CABAC

算术编码是一种高效的熵编码方案，其每个符号所对应的码长被认为是分数。由于对每一个符号的编码都与以前编码的结果有关，所以它考虑的是信源符号序列整体的概率特性，而不是单个符号的概率特性，因而它能够更大程度地逼近信源的极限香浓熵，降低码率。为了绕开算术编码中无限精度小数的表示问题及对信源符号概率进行估计，现代的算术编码多以有限状态机的方式实现，H.264/AVC 的 CABAC 便是一个例子，其他的例子还有 JPEG 2000。在 CABAC 中，每编码一个二进制符号，编码器就会自动调整对信源概率模型（用一个“状态”来表示）的估计，随后的二进制符号就在这个更新了的概率模型基础上进行编码。这样

的编码器不需要信源统计特性的先验知识，而是在编码过程中自适应地进行估计。显然，与CAVLC编码中预先设定好若干概率模型的方法比较起来，CABAC有更大的灵活性，可以获得更好的编码性能——大约降低10%的码率。

5．去除块效应环路滤波器

去除块效应滤波器（Deblocking Filter或Loop Filter），它的作用是消除经反量化和反变换后重建图像中由于预测误差产生的块效应，即块边缘处的像素值跳变，从而改善图像的主观质量及来减少预测误差。H.264/AVC中的Deblocking Filter还能够根据图像内容作出判断，只对由于块效应产生的像素值跳变进行平滑，而对图像中物体边缘处的像素值不连续给予保留，以免造成边缘模糊。与以往的Deblocking Filter不同的是，经过滤波后的图像将根据需要放在缓存中用于帧间预测，而不是仅仅在输出重建图像时用来改善主观质量，也就是说该滤波器位于解码环中而非解码环的输出外，因而它又称做环路滤波Loop Filter。需要注意的是，对于帧内预测，使用的是未经过滤波的重建图像。

6．多参考帧的帧间预测

以往的P帧编码以前面重建的帧作为参考，进行运动估计预测得到最佳匹配块。B帧编码同样是以前后重建帧为参考预测得到残差。H.264/AVC提供了多个参考帧的帧间预测模式，使估计到的宏块更贴近实际运动物体模型，运动向量表示更精确。但在选用多帧参考的同时，计算量巨增，在使用DSP硬件实现H.264/AVC时，多帧参考的帧间预测并不启用。

7．提供SP/SI帧支持

H.264/AVC在支持I/P/B帧编码方式的同时，增加支持SP/SI帧编码方式。以上3种与以前的标准相似，主要是参考图像不同，其余两种像条类型是SP（切换P）和SI（切换I），是新的类型，用于在不同位率编码码流之间进行有效切换。SP帧称做切换P帧，能在不同的编码图像之间有效地切换。SI帧称做切换I帧，允许SP帧的宏块完全匹配，达到随机读取数据进行解码和恢复错误的目的。

8．其他新特性

1）编码器控制

视频编码的控制是编码器实现的核心问题。由于视频序列中的图像内容随着空间与时间的不同而变化很大，需要为图像的不同部分选择不同的编码参数进行压缩编码，而编码控制的目的就是确定一组编码参数。H.264编码器采用基于Lagrangian优化算法的率失真优化模型实现视频编码的控制，其实现简单而且效率高。

2）参数集

H.264引入了参数集的概念，每个参数集包含了相应的编码图像的信息。序列参数集SPS包含的是针对一连续编码视频序列的参数，如帧数及POC的约束、参考帧数目、解码图像尺寸和帧场编码模式选择标识等。图像参数集PPS对应的是一个序列中某一幅图像或者某几幅图像，其参数如熵编码模式选择标识、片组数目、初始量化参数和去方块滤波系数调整标识等。

以上是H.264/AVC相对以往视频编码标准的技术主要的明显改进之处，效果也最明显。H.264/AVC相对MPEG-2/4、H.263，在相同码率下，其图像质量PSNR高出3dB多。图15-10

是标准视频测试序列 Foreman.qcif 的不同编码器的码率对 PSRN 的比较图。比较可以看出，H.264/AVC 的码率 Bit-rate 在相同质量 Quality（亮度 Y 的 PSNR）下，节省非常可观的码流。

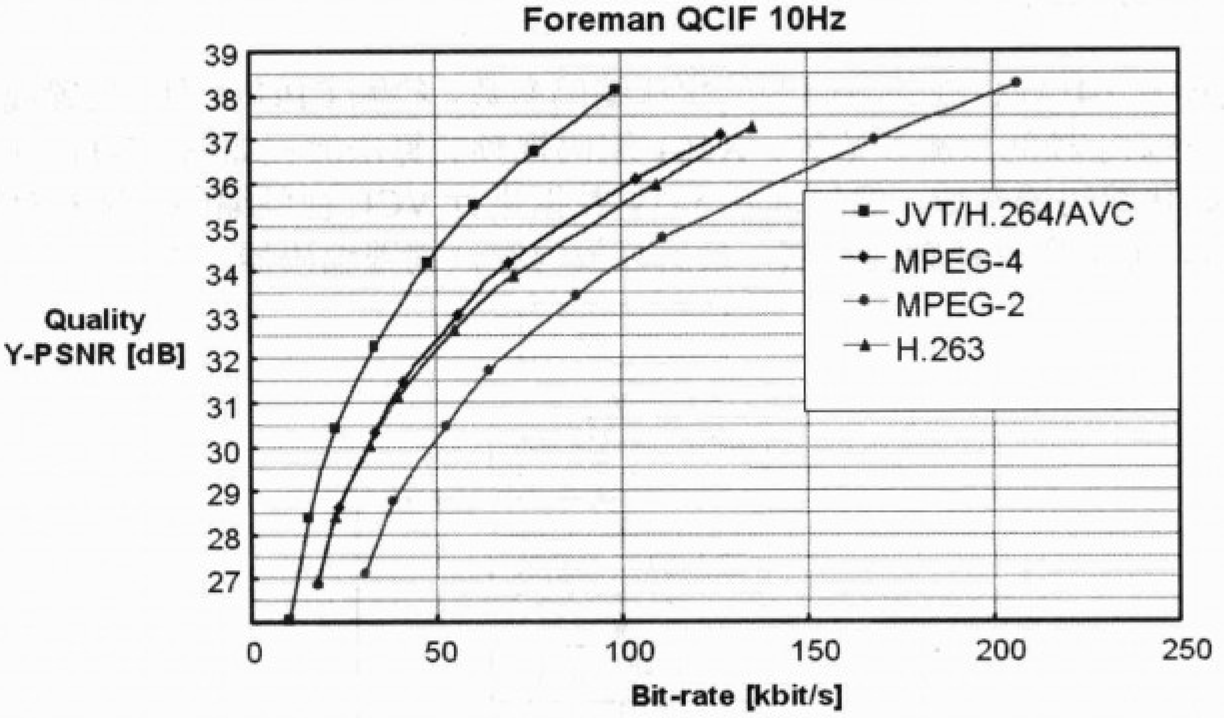


图 15-10 Foreman 序列的不同 CODEC 性能比较

表 15-1 不同 CODEC 相同图像质量的节省码流对比

codec	MPEG-4 ASP	H.263 CHC	MPEG-4 SP	H.263Base
H.264/AVC	28%	32%	34%	45%
MPEG-4 ASP	—	7%	10%	24%
H.264 CHC	—	—	—	18%
MPEG-4 SP	—	—	—	16%

表 15-1 列出了无 B 帧时相同图像质量 PSNR 条件下，平均节省码流百分比。分析可以看出，H.264/AVC 比 MPEG-4 SP 节省 34%的码流，这极大地鼓舞了人们大力进行研发投入。但是同时也应该看到，H.264/AVC 获得优越性能的代价是计算复杂度增加，据估计，编码的计算复杂度大约相当于 H.263 的 3 倍，解码复杂度大约相当于 H.263 的 2 倍。

不过让研发工程师欣慰的是，芯片 ASIC、可编程 DSP、FPGA、CPU 等实现平台的性能在日益提升，多核处理器也已出现，可编程的数字媒体处理器也已经出现，如 TI 的 DM64x 系列、ADI 的 Black Fin 系列、Philips 的 PNX 系列等，Intel 的双核 CPU 并支持 SSE4 媒体处理指令。硬件性能的逐步提升使实时实现复杂的视频算法已成为可能。

15.2.2 X264视频编码设计与剖析

X264 视频编码工程一直有更新，可以从网站上下载到最新的代码。这里以 X264-060308 版本为介绍对象，对其数据结构、算法数据流程和主要算法模块做剖析，根据 H.264/AVC 协议 T-REC-H.264-200305.doc 中的码流语义分析程序实现过程。

X264 工程包括 libx264 编码库和 X264 控制台应用程序。早期的 X264 工程可以使用 VC++ 6.0 编译，最新的 X264 需要 VC++ 7.0/8.0 来编译。后面的章节将详细介绍使用 VC++ 2005

（VC++ 8.0）运行 X264 的过程。

1．X264 应用工程

首先设置 H264/AVC 视频编码器的默认编码参数，检测 CPU 类型、设置视频属性、编码器参数、设置帧内/帧间预测参数等。X264 编码视频的算法流程如图 15-11 所示。首先创建编码器、初始化编码器参数，然后读入 YUV 数据进行 VCL 视频编码，接着对编码后的数据做 NAL 打包，循环所有视频视频帧，最后释放内存，销毁编码器。

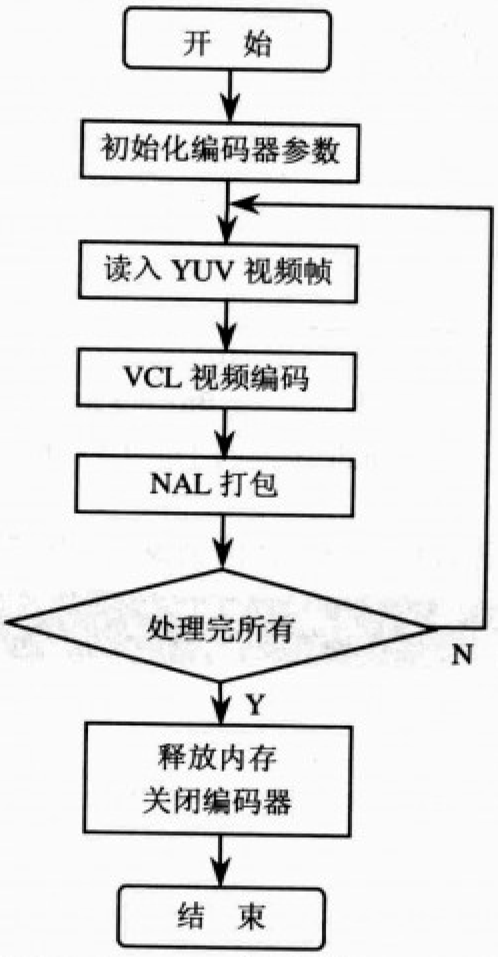


图 15-11　X264 视频编码流程图

下面详细介绍应用程序的数据结构和功能函数。

X264 的数据结构 x264_param_t 包含了 X264 编码器的所有参数字段。在编码前，X264 首先填充默认的参数集，同时检测 CPU 类型以使用 MMX 汇编函数初始化功能函数。填充默认参数集的函数为 x264_param_default。

```
/* x264_param_default：*用默认值填充x264_param_t，并检测PCU类型*/
void   x264_param_default( x264_param_t * );
```

由于 X264 应用程序是控制台类型，所以需要在 VC++的程序参数“Program arguments”中输入命令参数。Parse 函数读取输入的参数并剖析其合法性，然后初始化编码器参数 x264_param_t 并初始化读取 YUV 文件函数、写码流函数。

```
static int  Parse( int argc,            //参数个数
            char **argv,               //指向参数的指针
            x264_param_t *param,       //参数结构体
            cli_opt_t *opt );          //操作类型
```

最后根据设定的编码器参数开始对输入的视频文件进行 H.264/AVC 视频算法编码，对编码后的 H.264 码流写文件。

```
/* 编码一帧图像，然后NAL打包，最后写文件 */
static int  Encode_frame( x264_t *h, hnd_t hout, x264_picture_t *pic )
{
    x264_picture_t pic_out;                                    //待编码帧
    x264_nal_t *nal;                                           //NAL打包指针
    int i_nal, i;                                              //NAL包数目
    int i_file = 0;
    /* 编码一帧图像 */
    if( x264_encoder_encode( h, &nal, &i_nal, pic, &pic_out ) < 0 ) {
        fprintf( stderr, "x264 [error]: x264_encoder_encode failed\n" );
       //编码失败，输出当前信息
    }   for( i = 0; i < i_nal; i++ )                           //NAL层循环打包
    {
        int i_size;                                            //当前的打包空间大小
        if( mux_buffer_size < nal[i].i_payload * 3/2 + 4 ) //确保足够大的打包空间
        {
            mux_buffer_size = nal[i].i_payload * 2 + 4;        //2倍VCL长度+4
            x264_free( mux_buffer );              //释放前面申请的mux_buffer缓冲区
            mux_buffer = x264_malloc( mux_buffer_size );     //申请当前大小的缓冲区
        }
        i_size = mux_buffer_size;
        x264_nal_encode( mux_buffer, &i_size, 1, &nal[i] );//打包当前NAL层
        i_file += p_write_nalu( hout, mux_buffer, i_size );//写当前NAL层到文件
    }
    if (i_nal)
        p_set_eop( hout, &pic_out );   //存储当前帧编码码流结束标记eop
    return i_file;
}
/* 编码器 */
static int  Encode( x264_param_t *param, cli_opt_t *opt )
{
    x264_t *h;                              //编码器句柄
    x264_picture_t pic;                     //当前编码帧
    int       i_frame, i_frame_total;       //编码帧数目统计
    int64_t   i_start, i_end;               //编码时间统计
    int64_t   i_file;                       //当前NAL打包的长度
    int       i_frame_size;                 //编码后的码流长度
    int       i_update_interval;            //统计编码信息的间隔长度
    opt->b_progress &= param->i_log_level < X264_LOG_DEBUG;   //调试信息等级
    i_frame_total = p_get_frame_total( opt->hin );
                                            //获取视频文件的总编码帧数目
    i_frame_total -= opt->i_seek;
    if( ( i_frame_total == 0 || param->i_frame_total < i_frame_total )
        && param->i_frame_total > 0 )
        i_frame_total = param->i_frame_total;
   param->i_frame_total = i_frame_total;      //总编码帧数目
```

```
    //间隔1000帧统计编码信息
    i_update_interval = i_frame_total ? x264_clip3( i_frame_total / 1000, 1, 10 ) :
    10;
  if( ( h = x264_encoder_open( param ) ) == NULL )
                                                     //创建X264编码器并读入所有编码器参数
    {
       fprintf( stderr, "x264 [error]: x264_encoder_open failed\n" );
                                                            //创建编码器失败，打印信息
       p_close_infile( opt->hin );                          //关闭输入的YUV文件
       return -1;
    }
    if( p_set_outfile_param( opt->hout, param ) )        //设置写码流文件的参数
    {
       fprintf( stderr, "x264 [error]: can't set outfile param\n" );
                                                            //设置失败，打印信息
       p_close_infile( opt->hin );                          //关闭YUV文件
       p_close_outfile( opt->hout );                        //关闭码流文件
       return -1;
    }
    /* 创建一帧图像空间 */
    x264_picture_alloc( &pic, X264_CSP_I420, param->i_width, param->i_height );
    /* 读取当前系统时间 */
    i_start = x264_mdate();
    /* 循环编码所有视频帧 */
    for( i_frame = 0, i_file = 0; b_ctrl_c == 0 && (i_frame < i_frame_total ||
    i_frame_total == 0); )
    {
       /* 从YUV文件读取一帧视频图像 */
       if( p_read_frame( &pic, opt->hin, i_frame + opt->i_seek ) )
          break;
       pic.i_pts = (int64_t)i_frame * param->i_fps_den;
       if( opt->qpfile )                                           //量化文件
          parse_qpfile( opt, &pic, i_frame + opt->i_seek );    //分析量化文件
       else {
          /* 没有强制任何编码参数 */
          pic.i_type = X264_TYPE_AUTO;
          pic.i_qpplus1 = 0;
       }
       /* 编码一帧图像，h编码器句柄，hout存储码流文件，pic为预编码帧图像 */
       i_file += Encode_frame( h, opt->hout, &pic );
       i_frame++;        //编码帧数目递增
       /* 显示编码信息、当前状态等 */
       if( opt->b_progress && i_frame % i_update_interval == 0 )
          {                                                 //开启显示进度b_progess
          int64_t i_elapsed = x264_mdate() - i_start; //消耗时间
          double fps = i_elapsed > 0 ? i_frame * 1000000. / i_elapsed : 0;
```

```
                                                         //计算fps
        if( i_frame_total )
        {                                                //显示编码信息
            int eta = i_elapsed * (i_frame_total - i_frame) / ((int64_t)i_frame
            * 1000000);
         fprintf( stderr, "encoded frames: %d/%d (%.1f%%), %.2f fps, %.2f kb/s,
         eta %d:%02d:%02d  \r",
                i_frame, i_frame_total, 100. * i_frame / i_frame_total, fps,
            (double) i_file * 8 * param->i_fps_num / ( (double) param->i_fps_den
            * i_frame * 1000 ),
                eta/3600, (eta/60)%60, eta%60 );
        }
        else
            fprintf( stderr, "encoded frames: %d, %.2f fps   \r", i_frame, fps );
            fflush( stderr );           //刷新输出缓冲区
      }
    }
    /* 排空延迟的B帧 */
    do {
        i_file +=
        i_frame_size = Encode_frame( h, opt->hout, NULL );
    } while( i_frame_size );
    /* 获取系统当前时间 */
    i_end = x264_mdate();
    x264_picture_clean( &pic );         //释放申请的一帧图像空间
    /* 在打印编码状态前清除进程标识 */
    if( opt->b_progress )
fprintf(stderr,"          \r" );
    x264_encoder_close( h );            //关闭编码器
    x264_free( mux_buffer );            //释放内存
    fprintf( stderr, "\n" );
    if( b_ctrl_c )                      //编码过程中出错，输出编码帧序号
        fprintf( stderr, "aborted at input frame %d\n", opt->i_seek + i_frame );
    p_close_infile( opt->hin );         //关闭输入的YUV文件
    p_close_outfile( opt->hout );       //关闭输出的H.264文件
    if( i_frame > 0 ){                  //打印编码器的当前CPU编码统计信息：fps、kpbs
        double fps = (double)i_frame * (double)1000000 /
                  (double)( i_end - i_start );
        fprintf( stderr, "encoded %d frames, %.2f fps, %.2f kb/s\n", i_frame, fps,
              (double) i_file * 8 * param->i_fps_num /
              ( (double) param->i_fps_den * i_frame * 1000 ) );
    }
    return 0;
}
```

上述代码中的 Encode 为 X264 编码器，对 YUV 文件进行编码。首先函数 p_read_frame() 读取一帧图像，然后函数 Encode_frame()完成编码工作。编码中 x264_encoder_encode()编码

一帧图像实现 VCL，接着函数 x264_nal_encode()实现 VCL 打包，并存储码流到文件。循环调用 Encode_frame()实现 YUV 文件的 H.264 编码工作。

main 函数作为 X264 控制台应用程序的入口。使用上面的功能函数，完成 H.264 视频编码，代码如下。

```
int main( int argc, char **argv )
{
    x264_param_t param;                          //编码器参数
    cli_opt_t opt;                               //编码器操作
    int ret;
#ifdef _WIN32
    _setmode(_fileno(stdin), _O_BINARY);         //二进制格式
    _setmode(_fileno(stdout), _O_BINARY);        //二进制格式
#endif
    /* 编码器参数默认初始化 */
    x264_param_default( &param );
    /* 剖析输入的命令，初始化编码器 */
    if( Parse( argc, argv, &param, &opt ) < 0 )
        return -1;
    /* H.264视频编码 */
    ret = Encode( &param, &opt );
    return ret;
}
```

上述代码是 X264 编码器的主函数，首先初始化编码器的默认参数，然后剖析数据命令参数的有效性并初始化编码器，最后实现 H.264 视频编码。

2. libx264 编码库

X264 编码库 libx264 实现真正的视频帧编码，根据前面 H.264/AVC 关键技术分析，该视频编码算法仍然是基于块的混合编码技术，即帧内/帧间预测，对预测值变换、量化，最后是熵编码。编码帧类型仍然分为 I 帧（X264_TYPE_I）、P 帧（X264_TYPE_P）、B 帧（X264_TYPE_B），在 H.264 中叫图像片 Slice。X264 把一整帧看做一个片，片中有 SLICE_TYPE_I、SLICE_TYPE_P、SLICE_TYPE_B 之分。I 帧只有 SLICE_TYPE_I，P 帧有 SLICE_TYPE_I、SLICE_TYPE_P，B 帧 3 种片可以都有。

X264 的 H.264 视频编码过程大致可以分为 3 个过程，首先根据规则判定当前帧的编码类型，如果是 B 帧则还要缓冲存放、获取。然后对编码图像采用帧内预测、帧间预测、整数 DCT 变换、量化和熵编码。最后把压缩后的码流进行 NAL 层打包输出。

```
/************************************************************************
 * 视频帧结构和函数
 ***********************************************************************/
typedef struct
{
    int      i_csp;            /* 图像颜色空间，目前只支持I420/YUV420  */
    int      i_plane;          /* 平面数目，如I420是3  */
```

```
    int       i_stride[4];       /* 每个平面的列跨度 */
    uint8_t *plane[4];          /* 每个平面的指向空间 */
} x264_image_t;                 //待编码的图像
typedef struct
{
    /* In: 强制帧类型（如果不是自动判断）XXX: 现在忽略此选项
     * Out: 已经编码帧的编码类型*/
    int     i_type;
    /* In: 强制量化器 */
    int     i_qpplus1;
    /* In: 用户pts。 Out: 已编码帧的pts*/
    int64_t i_pts;
    /* In: 原始YUV视频帧数据 */
    x264_image_t img;
} x264_picture_t;               //X264编码视频帧
/* x264_picture_alloc: * 申请一帧图像空间，需要调用 x264_picture_clean 释放申请的
                        内存 */
void x264_picture_alloc( x264_picture_t *pic, int i_csp, int i_width, int i_
height );
/* x264_picture_clean:
 * 释放由 x264_picture_alloc 申请的有关资源
 * 视频结构体是x264_picture_t */
void x264_picture_clean( x264_picture_t *pic );
x264_t   *x264_encoder_open   ( x264_param_t * );
int      x264_encoder_headers( x264_t *, x264_nal_t **, int * );
int      x264_encoder_encode ( x264_t *, x264_nal_t **, int *, x264_picture_t * );
void     x264_encoder_close  ( x264_t * );
/****************************************************************************
 * NAL层打包
 ****************************************************************************/
typedef struct
{
    int i_ref_idc;              /* nal的优先级标志 */
    int i_type;                 /* nal数据类型: SPS/PPS等 */
    int     i_payload;          /* VCL层编码后的码流长度大小 */
    uint8_t *p_payload;         /*VCL层编码后的码流空间*/
} x264_nal_t;
/* x264_nal_encode:
 *       编码一个nal到缓冲区，设置大小
 *       如果b_annexeb为真，则需要增加长期的同步工作
 *       XXX: 现在没有检查溢出 */
int x264_nal_encode( void *, int *, int b_annexeb, x264_nal_t *nal );
```

上述代码是 X264 的编码器有关的数据结构和功能函数。x264_image_t 是实际参与编码的编码帧图像的信息，包括颜色空间、图像位平面数目、每个平面的宽度和图像平面地址指

针。x264_picture_t 是 X264 编码器定义的便于控制的图像帧，它包含了 x264_image_t 结构体。功能函数包括 VCL 编码和 NAL 打包，VCL 编码函数包括创建编码器 x264_encoder_open()、编码图像 x264_encoder_encode()、关闭编码器 x264_encoder_close()。函数 x264_nal_encode() 实现 NAL 打包。

3. 视频帧的存放与获取

X264 的编码器结构体 x264_t 中的子结构体字段 frames 包含了 4 个临时视频帧序列空间：current、next、unused 和 reference。分别保存当前编码帧、将编码帧序列、未处理原始视频帧序列和参考帧序列，同时 X264 编码器还申请了 fenc 和 fdec 空间用于存放已编码帧和重建帧。

注意：

H.264 算法中的“帧”与“片”都是图像帧，后面不加特别说明均指相同意义。

编码器处理视频帧的顺序如下。

STEP 01 从 YUV 视频文件中读取图像存储到临时 x264_picture_t 变量 pic_in，同时为 unused 申请存储空间，并用 fenc 指针指向这个空间。

STEP 02 将 pic_in 中的图片数据复制到 fenc 所指向的空间，并在复制完成后对图片大小进行判断，如果长宽不为 16 的整数倍则进行像素扩展；将处理后的 fenc 区域数据放入 next 区域。之后，如果存在 B 帧，则从 next 区域取出 B 帧以后的 P 帧放到 current 区域中，也就是说先编码 I、P 帧再编码之间的 B 帧；否则，直接从 next 区域取出一帧存入 current 区域。此时 current 区域中存放的就是经过预处理的即将要编码的帧数据。

STEP 03 由于 fenc 区域是编码的直接对象，再将 current 区域中的内容复制到 fenc 中正式开始编码。这个过程的程序实现在文件 Enc/encoder.c 中的 x264_encoder_encode()函数中，代码如下。

```
/* ------------------- 从编码器的图像序列中获取编码帧-------------------- */
TIMER_START( i_mtime_encode_frame );          //开始获取图像编码帧，记录当前系统时间
if( pic_in != NULL )
{
    /* 1: 复制图像到frame，并移动到buffer */
    x264_frame_t *fenc = x264_frame_get( h->frames.unused );
    x264_frame_copy_picture( h, fenc, pic_in );       //复制图像：fenc = pic_in;
    if( h->param.i_width % 16 || h->param.i_height % 16 )
                                              //图像宽或高不是16的倍数
    x264_frame_expand_border_mod16( h, fenc );
                                              //扩展图像边使其宽和高成为16的倍数
    fenc->i_frame = h->frames.i_input++;
    x264_frame_put( h->frames.next, fenc );   //next = fenc;
    /* 是否是低分辨率图像*/
    if( h->frames.b_have_lowres )
        x264_frame_init_lowres( h->param.cpu, fenc );//把fenc图像降低分辨率
    if( h->frames.i_input <= h->frames.i_delay )
    {
```

```
            /* 编码器没有输入，等待填充bframe缓冲 */
            pic_out->i_type = X264_TYPE_AUTO;
            return 0;
        }
    }
    if( h->frames.current[0] == NULL )          //以下为填充、获取B帧编码图像
    {
        int bframes = 0;
        /* 2: 选择帧类型 */
        if( h->frames.next[0] == NULL )
            return 0;
        /* 确定帧类型*/
        x264_slicetype_decide( h );
        /* 3: 把一些B帧移动到编码序列*/
        while( IS_X264_TYPE_B( h->frames.next[bframes]->i_type ) )
             bframes++;
        //帧next[bframes]赋给current帧
        x264_frame_put( h->frames.current, x264_frame_get( &h->frames.next
[bframes] ) );
        /* 注意:当B-frames>3时，在GOP关闭时BREF帧可能不再处于中间*/
        if( h->param.b_bframe_pyramid && bframes > 1 )
        {
             x264_frame_t *mid = x264_frame_get( &h->frames.next[bframes/2] );
             mid->i_type = X264_TYPE_BREF;   //帧类型为B帧的参考帧，不可任意处理
             x264_frame_put( h->frames.current, mid );   //中间帧赋给current
             bframes--;
        }
        while( bframes-- )
             x264_frame_put(h->frames.current,x264_frame_get(h->frames.next));
    }
    TIMER_STOP( i_mtime_encode_frame );        //获取编码帧的工作结束，记录当前系统时间
    /* ------------------ Get frame to be encoded ------------------------ */
    /* 4: 获取待编码图像 */
    h->fenc = x264_frame_get( h->frames.current );
    //从current中获取编码图像，赋给fenc
    if( h->fenc == NULL )
    {
        /* 编码器没有图像编码，如I/P帧等待B帧*/
        /*  等待填充bframe缓冲*/
        pic_out->i_type = X264_TYPE_AUTO;      //编码类型为系统自动判断
        return 0;
    }
```

上述代码实现编码器的图像序列填充、获取当前编码图像的过程。对于 B 帧图像，填充过程较为复杂。pic_in 为输入的图像，最后编码器获取的待编码图像是 fenc。

4. 视频帧压缩编码 VCL

在上述过程获取到编码图像后，编码器开始 H.264 视频编码 VCL。首先初始化有关参数，包括帧类型的获取、创建多参考帧的列表、初始化码流控制、初始化写码流结构和写 SPS、PPS 头结构信息。

1）初始化相关参数

若为 IDR（Instantaneous Decoder Refresh）帧，则意味着是一个新图像片的开始。在 H.264 中为了防止编码错误扩散，规定当前片不以本片以外的其他帧为参考，因而编码器遇到 IDR 帧时需要重置参考帧区域。同时，将 SEI（Sequence Enhancement Information）、SPS（Sequence Parameter Set）和 PPS（Picture Parameter Set）分别单独写入 NAL 单元。这 3 个参数集集合了编/解码的核心参数，直接关系解码端能否正常解码。如果不是 IDR 帧，则依据帧类型设定当前 NAL 单元的类型和图像片的类型。接着进行一系列初始化参数的操作，包括建立关于参考帧的 list0 和 list1，初始化码流控制得到量化步长、参考帧等相关信息，初始化写比特流环境变量等。

```
if( h->fenc->i_type == X264_TYPE_IDR )          //编码帧为IDR
{
   h->frames.i_last_idr = h->fenc->i_frame;  //更新上次的IDR
}
/* ------------------ 设置帧环境变量 ---------------------------- */
/* 5: 根据帧类型数据初始化*/
TIMER_START( i_mtime_encode_frame );            //记录系统时间
if( h->fenc->i_type == X264_TYPE_IDR )          //IDR帧
{
      /* 重排参考帧 */
      x264_reference_reset( h );
      i_nal_type    = NAL_SLICE_IDR;            //nal类型NAL_SLICE_IDR
      i_nal_ref_idc = NAL_PRIORITY_HIGHEST;     //nal_ref_idc标识高级别
      i_slice_type = SLICE_TYPE_I;              //I帧
}
else if( h->fenc->i_type == X264_TYPE_I )       //I帧
{
      i_nal_type    = NAL_SLICE;                //图像片
      i_nal_ref_idc = NAL_PRIORITY_HIGH;
      i_slice_type = SLICE_TYPE_I;              //I帧
}
else if( h->fenc->i_type == X264_TYPE_P )       //P帧
{
      i_nal_type    = NAL_SLICE;                //图像片
      i_nal_ref_idc = NAL_PRIORITY_HIGH;
      i_slice_type = SLICE_TYPE_P;              //P帧
}
else if( h->fenc->i_type == X264_TYPE_BREF ) //B帧参考
{
      i_nal_type    = NAL_SLICE;                     //图像片
```

```
        i_nal_ref_idc = NAL_PRIORITY_HIGH;
        i_slice_type = SLICE_TYPE_B;                    //B帧
}
else {    /* B 帧 */
        i_nal_type    = NAL_SLICE;                      //图像片
        i_nal_ref_idc = NAL_PRIORITY_DISPOSABLE;        //编码图像，从序列中取消
        i_slice_type = SLICE_TYPE_B;                    //B帧
}
//fenc为当前编码帧图像，fdec为当前帧的重建帧
h->fdec->i_poc =                                        //计算poc
h->fenc->i_poc = 2 * (h->fenc->i_frame - h->frames.i_last_idr);
h->fdec->i_type = h->fenc->i_type;                      //帧类型
h->fdec->i_frame = h->fenc->i_frame;                    //编码帧计数器
h->fenc->b_kept_as_ref =
h->fdec->b_kept_as_ref = i_nal_ref_idc != NAL_PRIORITY_DISPOSABLE;
/* ------------------        初始化 ---------------------------- */
/* 构建参考帧列表 0/1 */
x264_reference_build_list( h, h->fdec->i_poc, i_slice_type );
/* 初始化码流控制 */
x264_ratecontrol_start( h, i_slice_type, h->fenc->i_qpplus1 );
i_global_qp = x264_ratecontrol_qp( h );                 //得到当前帧的量化步长
pic_out->i_qpplus1 =                                    //强制量化步长
h->fdec->i_qpplus1 = i_global_qp + 1;                   //强制量化步长
if( i_slice_type == SLICE_TYPE_B )                      //B帧编码
        x264_macroblock_bipred_init( h );               //B帧双向预测初始化
if( h->fenc->b_kept_as_ref )
        h->i_frame_num++;
/* ----------------------- 创建片头 ---------------------- */
x264_slice_init( h, i_nal_type, i_slice_type, i_global_qp );
/* -----------------------   写码流 ---------------------- */
/* 初始化 bitstream 环境变量 */
h->out.i_nal = 0;
bs_init( &h->out.bs, h->out.p_bitstream, h->out.i_bitstream );
if(h->param.b_aud){                                     /* 使用用户单元定义符号 */
        int pic_type;
        if(i_slice_type == SLICE_TYPE_I)
                pic_type = 0;                           //I帧为0
        else if(i_slice_type == SLICE_TYPE_P)
                pic_type = 1;                           //P帧为1
        else if(i_slice_type == SLICE_TYPE_B)
                pic_type = 2;                           //B帧为2
        else
                pic_type = 7;
        x264_nal_start(h, NAL_AUD, NAL_PRIORITY_DISPOSABLE); //nal开始，整理nal
        bs_write(&h->out.bs, 3, pic_type);              //写用户单元定义符合
        bs_rbsp_trailing(&h->out.bs);                           //填充码流结尾
```

```
        x264_nal_end(h);                                    //nal结束，整理nal
    }
    h->i_nal_type = i_nal_type;                             //存储nal类型
    h->i_nal_ref_idc = i_nal_ref_idc;                       //存储 nal_ref_idc
    /* 写SPS和PPS参数集 */
    if( i_nal_type == NAL_SLICE_IDR && h->param.b_repeat_headers )
    {
        if( h->fenc->i_frame == 0 )
        {
            /* 标明用户自己 */
            x264_nal_start( h, NAL_SEI, NAL_PRIORITY_DISPOSABLE );
                                                            //开始nal，整理nal
            x264_sei_version_write( h, &h->out.bs );        //写sei信息
            x264_nal_end( h );                              //结束nal，整理nal
        }
        /* 产生序列参数集 */
        x264_nal_start( h, NAL_SPS, NAL_PRIORITY_HIGHEST );//开始nal，整理nal
        x264_sps_write( &h->out.bs, h->sps );              //写sps信息
        x264_nal_end( h );                                 //结束nal，整理nal
        /* 产生图像参数集 */
        x264_nal_start( h, NAL_PPS, NAL_PRIORITY_HIGHEST );//开始nal，整理nal
        x264_pps_write( &h->out.bs, h->pps );              //写pps信息
        x264_nal_end( h );                                 //结束nal，整理nal
    }
```

上述代码实现 VCL 的片编码初始化工作：帧类型的获取、写码流结构的初始化、码流控制获取当前帧编码量化步长、写片头序列参数集（SPS）、图像参数集（PPS）。写完头信息后，下面就是视频图像的实际编码。

2）以宏块为单位对视频数据进行编码

在开始编码前，写片头到码流信息中。如果同时使用了 CABAC 编码，则需要初始化 CABAC 环境变量。编码器在进行预测编码之前必须得到周围块的相关信息，通过 x264_macroblock_cache_load()函数将相关数据载入缓存，X264 开辟了 intra4x4_pred_mode 和 non_zero_count 区域用来存放当前宏块左侧和上侧相邻宏块的预测模式和非零值个数。开辟的 ref/mv/mvd 存储参考帧列表、运动向量和 cabac 编码用到的 mvd。

```
void x264_macroblock_cache_load( x264_t *h,        //编码器句柄
                                 int i_mb_x,       //当前编码宏块的x方向序号
                                 int i_mb_y );     //当前编码宏块的y方向序号
```

然后，利用 x264_macroblock_analyse()函数分析参数，确定最佳编码模式即宏块模式抉择，这部分是 H.264/AVC 的技术核心，有很多研究人员在这里发挥自己的才智，在保证编码效率的同时，设计和优化算法以降低运算量提高其实用性。I 帧从 9 种 4×4 模式和 4 种 16×16 模式中选择模式；P 帧选择帧间预测模式下的宏块分割方式或帧内预测模式下的 4×4 模式和 16×16 模式。P 帧的帧间预测模式从 16×16、16×8、8×16、8×8、8×4、4×8、4×4 等以 SAD 值决定宏块的最佳编码模式。

模式抉择函数 x264_macroblock_analyse()的实现过程如图 15-12 所示。首先初始化模式

抉择的参数和环境变量，接着判断帧编码类型，如果为 I 帧，则使用 x264_mb_analyse_intra()分析 I 帧的 Intra 宏块的编码模式，然后调用 x264_analyse_update_cache()存储模式抉择环境变量，完成模式抉择任务。如果为 P 帧，则首先判断该帧是否为跳过帧（不编码 not_coded）类型。如果需要编码，则使用帧间亮度快预测模式的几种宏块类型 x264_mb_analyse_inter_p16x16、x264_mb_analyse_inter_p8x8 等 7 种决定最佳预测模式，然后使用分数像素、Intra 模式选择最佳模式，接着分析色度的宏块编码模式。最后更新、存储模式抉择环境变量，完成模式抉择任务。这里所说的模式抉择相对以往的标注中的术语，对 I 帧来说就是帧内预测（Predict），对 P 帧来说就是运动估计 ME（Motion Estimation）。

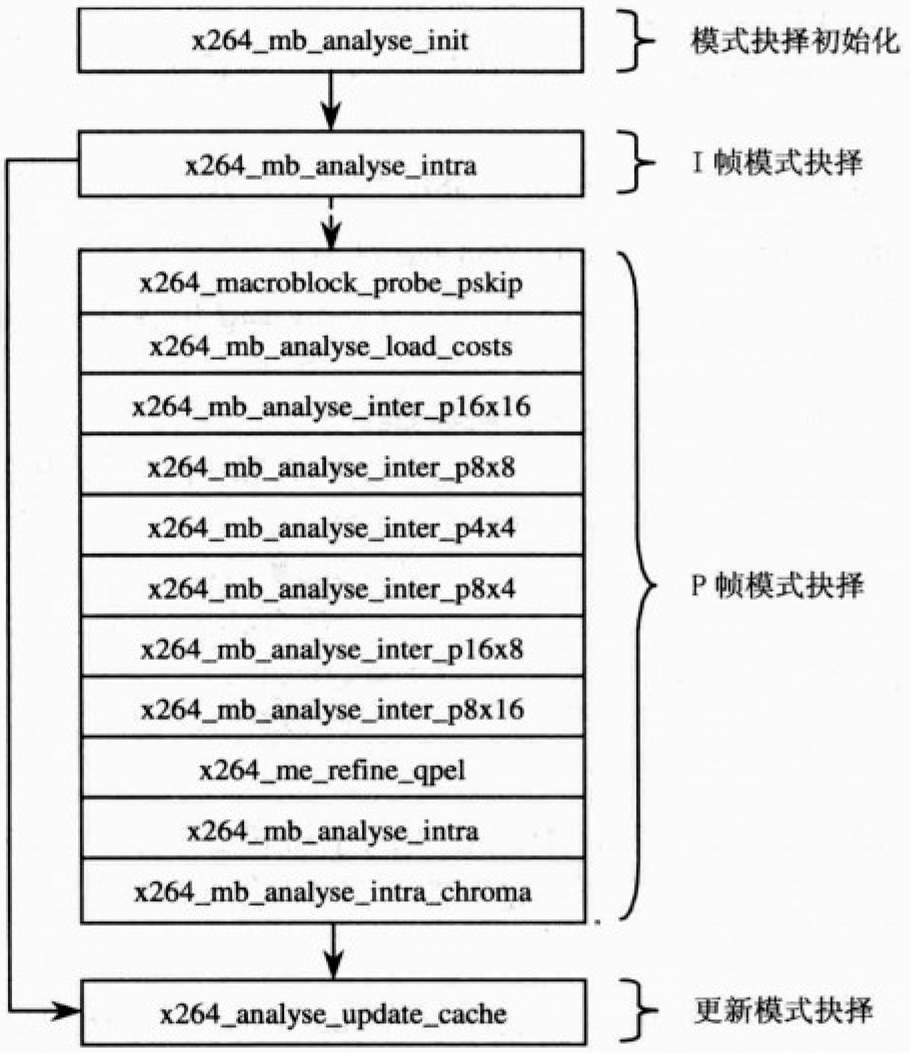


图 15-12　宏块模式抉择主要函数流程图

① void x264_macroblock_analyse(x264_t *h)

首先初始化函数 x264_mb_analyse_init()，然后根据帧类型使用不同的模式抉择函数。模式抉择的代码框架如下。

```
if( h->sh.i_type == SLICE_TYPE_I ){
        …….            //I帧模式抉择（帧内预测）
}else if( h->sh.i_type == SLICE_TYPE_P ){
        …….            //P帧模式抉择（运动估计）
}else if( h->sh.i_type == SLICE_TYPE_B ){
        ……             //B帧模式抉择（运动估计）
}
```

在上述过程的 P 帧模式抉择中，首先使用 Inter 块得到最佳模式，然后使用 I 帧的 Intra 块的模式抉择，综合得到最佳编码模式。

显然，对不同类型的宏块采用不同的处理技术，首先就 SLICE_TYPE_P 类型做深入分析，

P 类型包含有 I 模式的模式抉择，SLICE_TYPE_B 只是 SLICE_TYPE_P 的简单延伸而已。

```
if( h->param.analyse.b_fast_pskip )          //分析P帧是否是不编码模式（跳过编码）
{
if( h->param.analyse.i_subpel_refine >= 3 )  //模式抉择中分数像素精度>=3
 analysis.b_try_pskip = 1;                   //可以尝试判断跳过编码
else if( h->mb.i_mb_type_left == P_SKIP ||   //当前宏块的左、上
       h->mb.i_mb_type_top == P_SKIP ||      //上左、上右为跳过编码
       h->mb.i_mb_type_topleft == P_SKIP ||
       h->mb.i_mb_type_topright == P_SKIP )
 b_skip = x264_macroblock_probe_pskip( h );  //尝试跳过编码
}
```

条件进入后首先判断 skip 模式，如果满足 skip 模式则判断结束，否则继续下面的判断。

x264_mb_analyse_inter_p16x16(h, &analysis)函数是 P 帧模式抉择中 P 帧的必经模式。

```
if( flags & X264_ANALYSE_PSUB16x16 )          //宏块类型中可以使用16×16的子块
{
    if( h->param.analyse.b_mixed_references )
                                              //允许P帧中的每个宏块有自己的参考帧
        x264_mb_analyse_inter_p8x8_mixed_ref( h, &analysis );
                                              //8×8块模式抉择
    else                                      //P帧的宏块使用同一个参考帧
        x264_mb_analyse_inter_p8x8( h, &analysis );
                                              //8×8块模式抉择
}
```

上述代码首先判断编码参数中是否有 16×16 块的子块模式抉择。若选择使用则进一步判断，P 帧的宏块的参考帧是同一个参考帧还是有自己不同的参考帧。然后使用不同的函数完成 8×8 块的模式抉择。

分析 16×16 和 8×8 模式。模式代价值分别保存在 analysis.l0.me16x16.cost 和 analysis.l0.i_cost8x8 中。

```
    if( ( flags & X264_ANALYSE_PSUB16x16 ) && analysis.l0.i_cost8x8 < analysis.l0.
me16x16.cost )
    {
          //…….
    }
```

如果 8×8 代价值小于 16×16，则对 8×8 的宏块进一步分割，进行 4×4 块的判断。

```
for( i = 0; i < 4; i++ )                              //8×8块中有4个4×4的块
{
      x264_mb_analyse_inter_p4x4( h, &analysis, i );
                                                      //4×4块模式选择
      if( analysis.l0.i_cost4x4[i] < analysis.l0.me8x8[i].cost )
                                                      //4×4模式代价优于8×8
      {
            //………                                    //进一步分割：4×8、8×4
```

```
        }
}
```

依次对 4 个子宏块（8×8）进行处理，x264_mb_analyse_inter_p4x4()函数实际上是得到 4 个 4×4 块的代价和 analysis.l0.i_cost4x4[i]。如果 4×4 模式优于 8×8 模式，才进行 8×8 块的细分割。细分割代码分析略。

完成上述判断后，X264 编码器开始使用 16×8、8×16 的模式抉择，代码框架如下。

```
if( ( flags & X264_ANALYSE_PSUB16x16 ) &&
            analysis.l0.i_cost8x8 < analysis.l0.me16x16.cost + i_thresh16x8 )
{
        x264_mb_analyse_inter_p16x8( h, &analysis ); //选用宏块16×8模式抉择
        //……
        x264_mb_analyse_inter_p8x16( h, &analysis ); //选用宏块8×16模式抉择
        //……
}
```

上述代码是 X264 选用 16×8、8×16 的宏块方式的模式抉择。在 x264_mb_analyse_inter_p16x8 函数的内部，循环 2 次 16×8 宏块方式的模式抉择。在 x264_mb_analyse_inter_p8x16 函数的内部，同样要循环 2 次 8×16 宏块方式的模式抉择。

在检测 16×8 和 8×16 模式后，X264 编码器在帧间模式选择后，使用 x264_me_refine_qpel（h, &analysis.l0.me16x16）对当前最佳模式进行亚像素（分数像素）精细搜索，以进一步减少误差。值得注意的是，在前面每个模式的抉择中，也要进行亚像素搜索，见 x264_me_search_ref（）函数的最后几行，代码如下。

```
//.....................
/* 分数像素精细搜索 */
if( h->mb.i_subpel_refine >= 2 )
{
     int hpel = subpel_iterations[h->mb.i_subpel_refine][2]; //1/2像素精度
     int qpel = subpel_iterations[h->mb.i_subpel_refine][3]; //1/4像素精度
     refine_subpel( h, m, hpel, qpel, p_halfpel_thresh, 0 ); //分数像素精细搜索
}
```

实际上这里的亚像素搜索是在上述基础上再进行精细搜索的。二者亚像素搜索（包括半像素和 1/4 像素）的次数由 subpel_iterations[i][4]确定，而 i 由编译参数 subme 确定，看运行帮助：

```
-m, --subme <integer>        Subpixel motion estimation quality: 1=fast, 5=best
```

实际上 subme 就是决定模式选择前后亚像素估计的点数。subme 越大，压缩效率越好，计算量越大。

X264 编码器对 P 帧应用完 Inter 块的模式检测后，再使用 Intra 块的模式检测。然后将色度的判断标志和前面检测的结果进行比较，如果做色度检测，则使用 x264_mb_analyse_intra_chroma()函数，最后记录总的检测结果。

```
x264_mb_analyse_intra( h, &analysis, i_cost );     //运用Intra块模式抉择
if( h->mb.b_chroma_me && !analysis.b_mbrd &&       //色度块参与模式据则
    ( analysis.i_sad_i16x16 < i_cost               //Inter块模式结果优于Intra块
```

```
    || analysis.i_sad_i8x8 < i_cost
    || analysis.i_sad_i4x4 < i_cost ))
  {
    x264_mb_analyse_intra_chroma( h, &analysis );              //色度块进行模式抉择
    analysis.i_sad_i16x16 += analysis.i_sad_i8x8chroma;        //记录总的检测结果
    analysis.i_sad_i8x8 += analysis.i_sad_i8x8chroma;
    analysis.i_sad_i4x4 += analysis.i_sad_i8x8chroma;
  }
```

在上述代码中，分析宏块的帧内编码，包括亮度和色度。亮度的 16×16、8×8、4×4 代价分别存在 analysis.i_sad_i16x16、analysis.i_sad_i8x8、analysis.i_sad_i4x4 中，色度代价存在 analysis.i_sad_i8x8chroma 中。

至此，P 帧的模式抉择判断完成，现在比较得出最佳模式。对 P 帧来说，一般是绝大部分的 Inter 块和少量的 Intra 块，但代价值（SAD）会有区别。所以需要综合判断来决定块的类型和代价值。代码如下。

```
i_intra_type = I_16x16;                                  //默认初始化为Intra块的16×16
i_intra_cost = analysis.i_sad_i16x16;
if( analysis.i_sad_i8x8 < i_intra_cost )        {  //8×8模式最佳
    i_intra_type = I_8x8;                          //记录当前Intra块类型为8×8
    i_intra_cost = analysis.i_sad_i8x8;            //Intra块的代价（SAD）
}
if( analysis.i_sad_i4x4 < i_intra_cost )        {  //4×4模式最佳
    i_intra_type = I_4x4;                          //记录当前Intra块类型为4×4
    i_intra_cost = analysis.i_sad_i4x4;            //Intra块的代价（SAD）
}
/* 比较得到最佳的帧内预测模式*/
if( i_intra_cost < i_cost ) {          //i_cost为Inter块模式的代价
    i_type = i_intra_type;             //Intra块模式代价值小于Inter块模式代价值
    i_cost = i_intra_cost;             //以Intra块模式记录编码类型和代价值。
}
```

上述代码实现了 P 帧模式的最后选择。先初始化为 Intra 类型的 16×16，记录代价值。然后分别比较 Intra 块的 8×8、4×4，得到当前最佳。最后比较 Inter 块和 Intra 块的代价值，即帧内代价与帧间代价作比较，得到最佳编码模式及其代价值。

由于篇幅原因，B 帧的模式抉择不再详细说明。同 P 帧的模式抉择非常相似，只不过是 P 帧是前向预测的，而 B 帧是双向预测的。

模式抉择完毕后，还要记录当前模式抉择的环境变量，函数 x264_analyse_update_cache() 完成更新、记录。

X264 还实现了 RD（Rate Distortion）算法进行模式抉择的选项，如果支持宏块的 RD，则模式抉择用以下函数实现。

```
if( !analysis.b_mbrd )                   //b_mbrd=0
    x264_mb_analyse_transform( h );      //宏块RD算法模式选择
```

根据选项 b_mrd 是否支持宏块 RD 来进行模式抉择。

以上详细分析了模式抉择函数的工作过程，现在对其中的重要函数分别进行介绍。

② static inline int x264_macroblock_probe_pskip(x264_t *h)

该函数直接调用了 x264_macroblock_probe_skip(h, 0)。功能是探询宏块是否为跳过编码模式，部分重要代码分析如下。

```
if( !b_bidir )                          //P帧还是B帧，P=0,B=1
{                                       //P帧跳过模式
    //得到运行向量MV
    x264_mb_predict_mv_pskip( h, mvp );
    mvp[0] = x264_clip3( mvp[0], h->mb.mv_min[0], h->mb.mv_max[0] );
    mvp[1] = x264_clip3( mvp[1], h->mb.mv_min[1], h->mb.mv_max[1] );
    //运动补偿
    h->mc.mc_luma( h->mb.pic.p_fref[0][0], h->mb.pic.i_stride[0],
            h->mb.pic.p_fdec[0],    FDEC_STRIDE,
            mvp[0], mvp[1], 16, 16 );
}
    /* 先得到预测矢量MVp，然后对MVp进行饱和处理，再进行相应的运动补偿 */
    //得到亮度残差
   h->dctf.sub16x16_dct(dct4x4,h->mb.pic.p_fenc[0],FENC_STRIDE,  h->mb.pic.p_
   fdec[0], FDEC_STRIDE );
    for( i8x8 = 0, i_decimate_mb = 0; i8x8 < 4; i8x8++ )
    {
      /* 循环编码4x4块 */
      for( i4x4 = 0; i4x4 < 4; i4x4++ )
      {
         const int idx = i8x8 * 4 + i4x4;
         quant_4x4( h, dct4x4[idx], (int(*)[4][4])def_quant4_mf, i_qp, 0 );
         scan_zigzag_4x4full( dctscan, dct4x4[idx] );
         i_decimate_mb += x264_mb_decimate_score( dctscan, 16 );
         if( i_decimate_mb >= 6 )
         {
             /* 不作为P_SKIP */
             return 0;
         }
    }
}
```

上述代码实现了检测跳过编码模式，如果首先是 P 帧检测，则得到运动向量 MV 并进行饱和处理，再进行相应的运动补偿。接着得到亮度的残差，对 4 个 8×8 块进行变换、量化、Z 型扫描。即对每个 8×8 块中的 4 个 4×4 块进行量化、zigzag 扫描。最后得到 8×8 块的 i_decimate_mb 值。如果量化后系数中只有零星的非零系数，且都是 1 或–1，i_decimate_mb 就比较小。If（i_decimate_mb<6）可以将系数全变为 0。注意，其他模式下的残差编码也用到了该处理过程。程序后面是对色度进行处理，与亮度类似，不再进行讨论。

③ static void x264_mb_analyse_inter_p16x16(x264_t *h, x264_mb_analysis_t *a)

帧间预测的核心函数，和以往的视频编码的 16×16 宏块做运动估计技术类似，核心代码功能框架如下。

```
for( i_ref = 0; i_ref < h->i_ref0; i_ref++ )
{
    //循环搜索搜索每个参考帧
    x264_mb_predict_mv_16x16( h, 0, i_ref, m.mvp );
    //得到mvp
x264_mb_predict_mv_ref16x16( h, 0, i_ref, mvc, &i_mvc );
    //得到邻块的MV、前一帧对应位置的MV，可用来预测搜索起点，加速运动估计
x264_me_search_ref( h, &m, mvc, i_mvc, p_halfpel_thresh );
}
```

上述代码首先循环搜索每个参考帧读到的运动向量 mvp，通过对参考块的预测得到邻块的 MV、前一帧对应位置的 MV 可用来预测搜索起点，加速运动估计。调用函数 x264_me_search_ref()完成运动估计。

④ void x264_me_search_ref(x264_t *h, x264_me_t *m, int (*mvc)[2], int i_mvc, int *p_halfpel_thresh)

这里是实际进行运动估计的函数模块。

```
//参数、变量初始化
bmx = pmx = x264_clip3( ( m->mvp[0] + 2 ) >> 2, mv_x_min, mv_x_max );
                                     //运动向量嵌位
bmy = pmy = x264_clip3( ( m->mvp[1] + 2 ) >> 2, mv_y_min, mv_y_max );
                                     //运动向量嵌位
bcost = COST_MAX;
COST_MV( pmx, pmy );                 //分析MV，检测mvp
bcost -= p_cost_mvx[ bmx<<2 ] + p_cost_mvy[ bmy<<2 ];
for( i = 0; i < i_mvc; i++ )         //根据最初的值得到预测器
{
const int mx = x264_clip3( ( mvc[i][0] + 2 ) >> 2, mv_x_min, mv_x_max );
const int my = x264_clip3( ( mvc[i][1] + 2 ) >> 2, mv_y_min, mv_y_max );
if( mx != bmx || my != bmy )
   COST_MV( mx, my );                //分析MV，检测预测MV
}
COST_MV( 0, 0 );                     //分析MV，检测原点
switch (h->mb.i_me_method)           //根据运动估计方法做搜索估计
{
case  X264_ME_DIA:                   //diamond search, radius 1 (fast)
case  X264_ME_HEX:                   //hexagonal search, radius 2
case  X264_ME_UMH:                   //uneven multi-hexagon search
case  X264_ME_ESA:                   //exhaustive search (slow)
}
```

上述代码是 X264 的运动估计部分。首先参数变量初始化，先检测 mvp 点，再检测其他预测矢量，最后检测原点（0,0）。注意 mvp[0]保留的是 1/4 精度，所以除以 4 就变成了整像素精度。然后就是具体的搜索算法，搜索算法种类如下。

- 菱形搜索（Diamond Search，DIA）：用小菱形模板反复搜索，速度快。菱形算法还有大模板搜索，这里没用到。

- 六边形（Hexagonal Search，HEX）：先用六边形模板反复搜索，粗匹配。然后用小菱形模板搜索一次，得到最终的整像素运动矢量。
- 不均匀多六边形搜索（Uneven-cross Multi-Hexagon-grid Search，UMHexagonS）：基于 JM 的著名的运动估计算法。
- 连续消除法（Exhaustive Search，ESA）：全搜索法的运动估计，速度慢。

在整数像素运动估计后，要做亚像素精细搜索。而在整像素运动估计内部，同样也采用了亚像素运动估计。可根据亚像素的等级（i_subpel_refine）是否在内部来做亚像素运动估计。代码如下。

```
if( h->mb.i_subpel_refine >= 2 )                           //采用子像素的级别为2
{
    int hpel = subpel_iterations[h->mb.i_subpel_refine][2];    //1/2像素精度
    int qpel = subpel_iterations[h->mb.i_subpel_refine][3];    //1/4像素精度
    refine_subpel( h, m, hpel, qpel, p_halfpel_thresh, 0 );    //亚像素运动估计
}
```

上述代码实现了在整数像素内部做亚像素运动估计。亚像素搜索在 x264_macroblock_analyse()函数已经介绍过了。

⑤ void x264_me_refine_qpel(x264_t *h, x264_me_t *m)

该函数实现亚像素精细搜索。首先得到 1/2 像素、1/4 像素搜索的迭代次数（菱形小模板）分别为 hpel、qhpel，然后调用 refine_subpel()，该函数的部分核心代码如下。

```
/* 尝试预测MV的子像素部分 */
 if( hpel_iters )                                      //1/2迭代不为0
 {
     int mx = x264_clip3( m->mvp[0], h->mb.mv_min_spel[0], h->mb.mv_max_spel[0] );
                                                       //mx嵌位
     int my = x264_clip3( m->mvp[1], h->mb.mv_min_spel[1], h->mb.mv_max_spel[1] );
                                                       //my嵌位
     if( mx != bmx || my != bmy )
         COST_MV_SAD( mx, my, -1 );                    //分析mx、my，使用SAD
 }

/* 1/2像素搜索 */
bdir = -1;
for( i = hpel_iters; i > 0; i-- ) {                    //循环迭代1/2像素
      odir = bdir;
      omx = bmx;
      omy = bmy;
      COST_MV_SAD( omx, omy - 2, 0 );         COST_MV_SAD( omx, omy + 2, 1 );
                                              //小菱形
     COST_MV_SAD( omx - 2, omy, 2 );          COST_MV_SAD( omx + 2, omy, 3 );
                                              //小菱形
     if( bmx == omx && bmy == omy )
```

```
            break;
    }

    if( !b_refine_qpel )                                    //1/4迭代次数为0
    {
        bcost = COST_MAX;
        COST_MV_SATD( bmx, bmy, -1 );                       //分析当前的bmx、bmy，使用SATD
    }

    /* 在检测多参考帧时提前终止 */
    if( p_halfpel_thresh ) {
        if( (bcost*7)>>3 > *p_halfpel_thresh ) {
            m->cost = bcost;
            m->mv[0] = bmx;
            m->mv[1] = bmy;
            //不必检测mv
            return;
        }
        else if( bcost < *p_halfpel_thresh )
            *p_halfpel_thresh = bcost;
    }
    /*1/4搜索 */
    bdir = -1;
    for( i = qpel_iters; i > 0; i-- ) {                     //1/4像素迭代
        odir = bdir;
        omx = bmx;
        omy = bmy;
        COST_MV_SATD( omx, omy - 1, 0 );  COST_MV_SATD( omx, omy + 1, 1 );
                                                            //小模板
        COST_MV_SATD( omx - 1, omy, 2 );  COST_MV_SATD( omx + 1, omy, 3 );
                                                            //小模板
        if( bmx == omx && bmy == omy )
            break;
    }
    m->cost = bcost;                                        //记录代价值
    m->mv[0] = bmx;                                         //记录最佳mx
    m->mv[1] = bmy;                                         //记录最佳my
    m->cost_mv = p_cost_mvx[ bmx ] + p_cost_mvy[ bmy ];
```

对半像素精度进行 hpel_iters 次小菱形搜索。后面有 1/4 像素精度的 qpel_iters 次小模板搜索。

⑥ static void x264_mb_analyse_intra(x264_t *h, x264_mb_analysis_t *a, int i_cost_inter)

Intra 宏块的模式抉择依次检测 Intra_16x16、Intra_4x4、Intra_8x8 的最佳模式。i_satd_thresh

阈值根据帧类型进行设置。另外对于 Intra4x4、Intra8x8 要进行 DCT 变换、量化、反变换、反量化和重建等，为后续的块做参考。由于篇幅原因，这里不再详细讲述，具体代码请参见光盘。

⑦ static inline void x264_mb_analyse_transform（x264_t *h）

函数对残差进行 8×8 的 satd 变换，比较绝对和值，值较小对应的尺寸用于变换的尺寸。该模式只有选择了 RD 算法的宏块检测时才使用。由于篇幅原因，这里不再详细讲述，具体代码请参见光盘。

以上是宏块的模式抉择，帧内 Intra 和帧间 Inter 块应用不同的预测方法，综合判断以得到宏块的最佳编码模式，接下来是宏块的编码。

⑧ 宏块编码 void x264_macroblock_encode（x264_t *h）

函数 x264_macroblock_encode()完成宏块编码。编码过程严格按照编码框架中规定的 DCT 变换、zigzag 扫描和熵编码的过程以先亮度块再色度块的顺序进行。需要说明的是，编码器过程对 DC 参数和 AC 参数的处理是分开进行的：首先对整个宏块进行 DCT 变换，然后将亮度块和色度块的 DC 参数抽出分别组成 4×4 和 2×2 的子块并对剩余 AC 参数进行量化扫描，接着对 DC 子块进行 Hadmard 变换、量化、zigzag 扫描、反变换、反量化，再对剩下的 AC 参数进行反量化，最后将处理后的 DC 子块和 AC 子块重新组合并对组合后的宏块进行 IDCT 变换。

编码过程：先 P 帧、B 帧的 SKIP 判断和编码并返回。若为 I 帧，则依次按照 16×16、8×8、4×4 的编码，若为 Inter MB，则进行 4×4 编码，最后是色度块的编码。这里的编码是 DCT 变换、量化、反量化和重建，并统计量化后的非零系数数目，如图 15-13 所示。

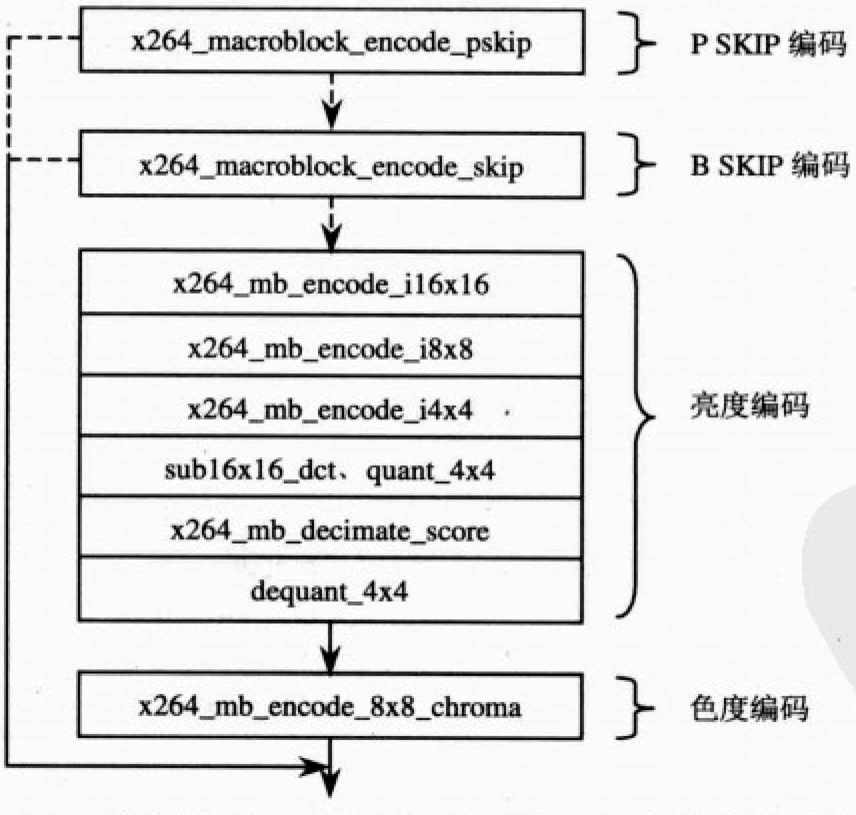


图 15-13　x264_macroblock_encode 模块流程图

在以上过程中，还需要统计 CBP（Coded Block Pattern）信息。CBP 一共 6bit，用于表示当前宏块是否存在非零值，主要用于决定熵编码时采用的码表。其中，高 2bit 表示 2 个 8×8 色度块的 CBP（值为 2 表示 U、V 中至少有一个 4×4 块的 AC 系数不全为 0；值为 1 表示 U、V 中至少有一个 2×2 的 DC 系数不全为 0；值为 0 表示所有色度系数全为 0）。低 4bit 分别对

应4个8×8亮度块，第i位（i=0、1、2、3）表示一个MB中第i行的4个4×4子块是否有非零参数。若存在非零值，则将相应位置1。将各子块的非零值个数保存到non_zero_count区域。使用先前得到的CBP查相应码表进行熵编码，将编码参数和编码数据写入比特流。

⑨ void x264_macroblock_write_cavlc（x264_t *h, bs_t *s）;

熵编码实质是查表编码。经变换、量化和重排后的系数根据CBP决定是否编码。编码内容包括当前宏块编码类型type、帧内块的预测类型、运动向量MV、编码块模式CBP等，然后编码残差系数residual：DC luma、AC Luma、Chroma DC residual等。详细熵编码过程见libx264工程，这里不再展开介绍。

⑩ void x264_macroblock_write_cabac（x264_t *h, x264_cabac_t *cb）;

CAVLC是Huffman编码，H.264中的熵编码还采用了CABAC（上下文自适应的二进制算法编码），编码效率更高。编码内容和CAVLC相同，只是编码的查表内容不同。由于篇幅原因，这里不再展开介绍，详细过程参见光盘。

3）保存编码信息

利用x264_macroblock_cache_save()函数将当前MB下一行和右一列的子块的编码模式保存到intra4x4_pred_mode，将子块非零系数的数目保存到non_zero_count中，为下一次预测编码采用。

4）编码器后处理

在完成一帧图像的编码后，去除重建图像的块效应，作环路滤波；扩展图像边；视频图像三方向v、h、hv插值；插值图像的边扩展；更新参考序列图像。回复CPU状态，设置输出视频帧的属性如编码类型，设置码流控制器状态。最后，统计MB信息，更新参考帧列表，并从unused区域取出新的帧放入fenc区域以备下次编码使用，打印编码器编码统计信息。

5. 码流NAL打包输出

上述的视频压缩编码对整幅图像的所有宏块循环完毕，实现H.264/AVC的视频VCL编码。NAL打包是为了增强码流的鲁棒性，适应网络传输。对VCL编码数据加上NAL头信息组成完整的NAL单元，输出到本地文件。

```
/****************************************************************************
 * x264_nal_encode:
 ****************************************************************************/
int x264_nal_encode( void *p_data,          //目标地址
                     int *pi_data,          //目标空间长度
                     int b_annexeb,         //是否有开始码
                     x264_nal_t *nal )      //待打包的数据
{
    uint8_t *dst = p_data;                  //目标地址
    uint8_t *src = nal->p_payload;          //待打包数据的起始地址
    uint8_t *end = &nal->p_payload[nal->i_payload]; //待打包数据的结束地址
    int i_count = 0;                                //0值计数器
    /* 注意：本程序没有检查溢出*/
    if( b_annexeb ) {
        /* long型nal开始码 (we always use long ones)*/
```

```
        *dst++ = 0x00;        *dst++ = 0x00;
       *dst++ = 0x00;        *dst++ = 0x01;
    }
    /* nal header */
    *dst++=(0x00<<7)|(nal->i_ref_idc<<5)|nal->i_type; //存储 ref_idc|type
    while( src < end ) {                                    //把所有内容打包
        if( i_count == 2 && *src <= 0x03 )
        {                                   //前面已经有连续2个0，同时当前值小于0x03
            *dst++ = 0x03;                  //则把当前值修改为0x03
            i_count = 0;                    //0值计数器清零
        }
        if( *src == 0 )  i_count++;         //当前值为0则计数器加1
        else        i_count = 0;            //否则清零
        *dst++ = *src++;                    //存储当前值
    }
    *pi_data = dst - (uint8_t*)p_data;      //打包后的数据长度
    return *pi_data;
}
```

上述代码实现 NAL 打包，NAL 头是固定的 0x00 00 00 01 标志，紧接着是 ref_idc 和 type 值。然后对要打包的数据循环，规则是若前面已经有连续两个 0 同时当前值小于 0x03，则把当前值修定为 0x03 并存储，0 值计数器清零。当前值为 0 计数器 count 加 1，否则 count 清零。存储当前值，最后得到打包后的数据长度。

以上分析了 X264 视频编码过程，对编码框架有了较为清楚的认识。要特别说明的是，在研究 X264 的过程中，设置断点调试，并根据协议文档对照来理解算法的具体工程实现，是一个基本的学习过程。前述介绍的是视频算法 H.264/AVC 的编码框架，底层的函数模块如 DCT/IDCT、量化/反量化、SAD/SATD、运动补偿插值等没有介绍。X264 是 PC 平台下的 H.264 工程实现，并且最新的 X264 版本提供了丰富的核心函数的 MMX/SSE2 汇编优化。DSP 平台没有提供可编译工程。

15.2.3 X264核心模块MMX/DSP汇编优化

X264 的底层核心模块，即处理视频数据的最底层模块。X264 的\common\x86 目录下提供了部分核心函数的 MMX/SSE2 的汇编优化、ADM64 位优化等。这些核心模块包括量化/反量化、DCT/IDCT、帧内预测 preditc、SAD/SATD、subdct/addidct、MC 插值、滤波 deblock 等。程序根据当前 CPU 的型号选择合适的汇编函数。

1）MMX/SSE2 汇编优化

```
x264_t *x264_encoder_open( x264_param_t *param )
{
   //..................
x264_predict_16x16_init( h->param.cpu, h->predict_16x16 );
                                        /* I帧亮度的16×16预测模式 */
   x264_predict_8x8c_init( h->param.cpu, h->predict_8x8c );
                                        /* I帧色度的8×8预测模式*/
```

```
    x264_predict_8x8_init( h->param.cpu, h->predict_8x8 );
                                                /* I帧亮度的8×8预测模式*/
    x264_predict_4x4_init( h->param.cpu, h->predict_4x4 );
                                                /* I帧亮度的4×4预测模式*/
    x264_pixel_init( h->param.cpu, &h->pixf );/*SAD/SATD模块的汇编初始化*/
    x264_dct_init( h->param.cpu, &h->dctf );  /*各种块的subDCT/IDCTadd */
    x264_zigzag_init( h->param.cpu, &h->zigzagf, h->param.b_interlaced );
                                                /*zigzag扫描汇编初始化*/
    x264_mc_init( h->param.cpu, &h->mc );      /*运动补偿、插值汇编初始化*/
    x264_quant_init( h, h->param.cpu, &h->quantf );
                                                /*各种块的quant/iquant汇编初始化*/
  x264_deblock_init( h->param.cpu, &h->loopf );/*去除块效应滤波的汇编初始化*/
  }
```

上述代码是根据 CPU 的型号对 X264 的底层核心函数指针使用汇编函数初始化。包括 I 帧的亮度、色度预测模式，SAD/SATD 模块、DCT/IDCT、Z 型扫描、运动补偿 MC、量化/反量化、去除块效应滤波器等。核心模块的 MMX/SSE2 汇编程序及初始化详见光盘。

2）DM642 线性汇编优化

X264 以其清晰的数据结构、兼容标准 H.264 码流作为蓝本被移植到其他平台。DSP 可编程实现视频编码是普遍的首选方案。DM642 已经被视频监控、数字媒体处理算法应用等证明是一款优秀的方案。DM642 的视频编/解码相关技术在前面的章节已经作了详细的介绍。这里我们使用其简单、快捷、高效的算法优化方法：使用 Intrinsic 指令优化核心函数。尽管有极少的汇编指令没有对应的 Intrinsic 指令，而使用了 C 语言实现，这基本不影响算法的优化效率。另外，使用 Intrinsic 指令的 C 语言环境编译和调试，给算法工程师提供了友好的开发环境，不像使用线性指令汇编、并行指令汇编，变量查看不方便、调试信息少。

在 H.264 的编码算法中，宏块类型尽管很丰富，但是由于 DSP 的 CPU 资源有限，不可能实现所有宏块类型的编码。在 I 帧的宏块类型中，4×4 宏块类型可以不选用，而只选择 16×16 宏块类型，从而降低计算量。由此，调用函数 sub4x4_dc 的 sub8x8_dct 就可以直接采用 8×8 的宏块类型，16×16 宏块再调用 4 次 sub8x8_dct。

```
/*当前编码图像和参考帧图像作减法，得到残差*/
static inline void pixel_sub_wxh( int16_t *diff,       //目标残差
                                  int i_size,           //块宽、高
                                  uint8_t *pix1,        //图像块1地址
                                  int i_pix1,           //图像块1次行跨度
                                  uint8_t *pix2,        //图像块2地址
                                  int i_pix2 )          //图像块2次行跨度
{
    int y, x;
    for( y = 0; y < i_size; y++ ) {                     //图像块高
        for( x = 0; x < i_size; x++ ) {                 //图像块宽
            diff[x + y*i_size] = pix1[x] - pix2[x];     //相减，存储
        }
        pix1 += i_pix1;                                 //指向下一行
        pix2 += i_pix2;                                 //指向下一行
```

```
    }
}
/* 当前图像和参考图像作4x4减法，然后作整数DCT */
static void sub4x4_dct( int16_t dct[4][4], uint8_t *pix1, uint8_t *pix2 )
{
    int16_t d[4][4];
    int16_t tmp[4][4];
    int i;
    /*源和参考帧作减，得到图像残差*/
    pixel_sub_wxh( (int16_t*)d, 4, pix1, FENC_STRIDE, pix2, FDEC_STRIDE );
    //行DCT
    for( i = 0; i < 4; i++ )
    {
        const int s03 = d[i][0] + d[i][3];
        const int s12 = d[i][1] + d[i][2];
        const int d03 = d[i][0] - d[i][3];
        const int d12 = d[i][1] - d[i][2];
        tmp[0][i] =   s03 +   s12;
        tmp[1][i] = 2*d03 +   d12;
        tmp[2][i] =   s03 -   s12;
        tmp[3][i] =   d03 - 2*d12;
    }
    //列DCT
    for( i = 0; i < 4; i++ )
    {
        const int s03 = tmp[i][0] + tmp[i][3];
        const int s12 = tmp[i][1] + tmp[i][2];
        const int d03 = tmp[i][0] - tmp[i][3];
        const int d12 = tmp[i][1] - tmp[i][2];
        dct[i][0] =   s03 +   s12;
        dct[i][1] = 2*d03 +   d12;
        dct[i][2] =   s03 -   s12;
        dct[i][3] =   d03 - 2*d12;
    }
}
/*当前帧和参考帧作8x8的整数DCT*/
static void sub8x8_dct( int16_t dct[4][4][4], uint8_t *pix1, uint8_t *pix2 )
{
    sub4x4_dct( dct[0], &pix1[0], &pix2[0] );
    sub4x4_dct( dct[1], &pix1[4], &pix2[4] );
    sub4x4_dct( dct[2], &pix1[4*FENC_STRIDE+0], &pix2[4*FDEC_STRIDE+0] );
    sub4x4_dct( dct[3], &pix1[4*FENC_STRIDE+4], &pix2[4*FDEC_STRIDE+4] );
}
/*当前帧和参考帧作16x16的整数DCT*/
static void sub16x16_dct( int16_t dct[16][4][4], uint8_t *pix1, uint8_t *pix2 )
{
```

```
        sub8x8_dct( &dct[ 0], &pix1[0], &pix2[0] );
        sub8x8_dct( &dct[ 4], &pix1[8], &pix2[8] );
        sub8x8_dct( &dct[ 8], &pix1[8*FENC_STRIDE+0], &pix2[8*FDEC_STRIDE+0] );
        sub8x8_dct( &dct[12], &pix1[8*FENC_STRIDE+8], &pix2[8*FDEC_STRIDE+8] );
    }
```

上述代码是 X264 视频算法的 DCT 模块，包括 16×16、8×8、4×4 块。具体实现函数为 4×4，即 H.264 的 DCT 变换是 4×4 变换。8×8 块调用 4 次 4×4 的 DCT，16×16 块调用 4 次 8×8 的 DCT。

通过第 13 章的介绍，使用 DM642 优化 H.264/AVC 算法，其核心技术是：使循环建立流水，并行更多的指令；使 CPU 访问的数据/程序放置在片上 L1/L2。循环内有函数调用编译器无法建立流水。所以在禁用 I 帧的 4×4 块模式下，把 sub8x8_dct 中的 4 个 sub4x4_dct 函数调用去除，直接处理 8×8 图像块，优化过程如下。

```
    static void sub8x8_dct( int16_t dct1[4][4][4], uint8_t *restrict pix1, int i_pix1,
uint8_t *restrict pix2, int i_pix2 )
    {
     int x,y;
     int i4x4;
     uint32_t pix1_0_3210,pix1_1_3210,pix1_2_3210,pix1_3_3210;
     uint32_t pix2_0_3210,pix2_1_3210,pix2_2_3210,pix2_3_3210;
     uint32_t pix1_0_32, pix1_0_10,pix2_0_32, pix2_0_10,pix1_1_32, pix1_1_10,pix2_
     1_32, pix2_1_10;
     uint32_t pix1_2_32, pix1_2_10,pix2_2_32, pix2_2_10,pix1_3_32, pix1_3_10,pix2_
     3_32, pix2_3_10;
     uint32_t diff_0_32,diff_0_10;
     uint32_t diff_1_32,diff_1_10;
     uint32_t diff_2_32,diff_2_10;
     uint32_t diff_3_32,diff_3_10;
     uint32_t tmp_0_32,tmp_0_01;
     uint32_t tmp_1_32,tmp_1_01;
     uint32_t tmp_2_32,tmp_2_01;
     uint32_t tmp_3_32,tmp_3_01;
     uint32_t s_0_03_12,s_1_03_12,s_2_03_12,s_3_03_12;
     uint32_t d_0_03_12,d_1_03_12,d_2_03_12,d_3_03_12;
     const uint32_t mask_add1_add1_0=0x00010001,mask_add2_add1_0=0x00020001;
     const uint32_t mask_add1_sub1_0=0x0001ffff,mask_add1_sub2_0=0x0001fffe;
     const uint32_t mask_add1_add1_1=0x00010001,mask_add2_add1_1=0x00020001;
     const uint32_t mask_add1_sub1_1=0x0001ffff,mask_add1_sub2_1=0x0001fffe;
     uint32_t dct_0_0,dct_1_0,dct_2_0,dct_3_0;
     uint32_t dct_0_1,dct_1_1,dct_2_1,dct_3_1;
     uint32_t dct_0_2,dct_1_2,dct_2_2,dct_3_2;
     uint32_t dct_0_3,dct_1_3,dct_2_3,dct_3_3;
     uint32_t dct_0_32,dct_0_10,dct_1_32,dct_1_10;
     uint32_t dct_2_32,dct_2_10,dct_3_32,dct_3_10;
     uint32_t diff_0_01,diff_1_01,diff_2_01,diff_3_01;
```

```
uint32_t tmp_0_0,tmp_1_0,tmp_2_0,tmp_3_0;
uint32_t tmp_0_1,tmp_1_1,tmp_2_1,tmp_3_1;
uint32_t tmp_0_2,tmp_1_2,tmp_2_2,tmp_3_2;
uint32_t tmp_0_3,tmp_1_3,tmp_2_3,tmp_3_3;
uint8_t *restrict pix1_m,*restrict pix2_m;
int16_t *restrict dct;
#pragma MUST_ITERATE(4,4,4)                          //强制循环迭代次数
for (i4x4=0; i4x4<4; i4x4++)
{
    y = i4x4>>1;                                     //得到y方向索引
    x = i4x4&0x01;                                   //得到x方向索引
    pix1_m = pix1 + y*4*i_pix1 + x*4;                //得到当前图像的地址
    pix2_m = pix2 + y*4*i_pix2 + x*4;                //得到参考图像的地址
    dct = (int16_t *)(dct1[i4x4]);
    pix1_0_3210 = _amem4(&pix1_m[0*i_pix1]);         //4字节对齐读取
    pix1_1_3210 = _amem4(&pix1_m[1*i_pix1]);
    pix1_2_3210 = _amem4(&pix1_m[2*i_pix1]);
    pix1_3_3210 = _amem4(&pix1_m[3*i_pix1]);
    pix2_0_3210 = _amem4(&pix2_m[0*i_pix2]);
    pix2_1_3210 = _amem4(&pix2_m[1*i_pix2]);
    pix2_2_3210 = _amem4(&pix2_m[2*i_pix2]);
    pix2_3_3210 = _amem4(&pix2_m[3*i_pix2]);
    pix1_0_32 = _unpkhu4(pix1_0_3210); pix1_0_10 = _unpklu4(pix1_0_3210);
                                                     //拆包数据
    pix2_0_32 = _unpkhu4(pix2_0_3210); pix2_0_10 = _unpklu4(pix2_0_3210);
    pix1_1_32 = _unpkhu4(pix1_1_3210); pix1_1_10 = _unpklu4(pix1_1_3210);
    pix2_1_32 = _unpkhu4(pix2_1_3210); pix2_1_10 = _unpklu4(pix2_1_3210);
    pix1_2_32 = _unpkhu4(pix1_2_3210); pix1_2_10 = _unpklu4(pix1_2_3210);
    pix2_2_32 = _unpkhu4(pix2_2_3210); pix2_2_10 = _unpklu4(pix2_2_3210);
    pix1_3_32 = _unpkhu4(pix1_3_3210); pix1_3_10 = _unpklu4(pix1_3_3210);
    pix2_3_32 = _unpkhu4(pix2_3_3210); pix2_3_10 = _unpklu4(pix2_3_3210);
    diff_0_32 = _sub2(pix1_0_32,pix2_0_32);          //双16比特减
    diff_0_10 = _sub2(pix1_0_10,pix2_0_10);
    diff_1_32 = _sub2(pix1_1_32,pix2_1_32);
    diff_1_10 = _sub2(pix1_1_10,pix2_1_10);
    diff_2_32 = _sub2(pix1_2_32,pix2_2_32);
    diff_2_10 = _sub2(pix1_2_10,pix2_2_10);
    diff_3_32 = _sub2(pix1_3_32,pix2_3_32);
    diff_3_10 = _sub2(pix1_3_10,pix2_3_10);

    diff_0_01 = _packlh2(diff_0_10,diff_0_10); //数据打包
    diff_1_01 = _packlh2(diff_1_10,diff_1_10);
    diff_2_01 = _packlh2(diff_2_10,diff_2_10);
    diff_3_01 = _packlh2(diff_3_10,diff_3_10);
    //0 column of tmp
    s_0_03_12 = _add2(diff_0_01,diff_0_32);              //双16比特加
```

```
d_0_03_12 = _sub2(diff_0_01,diff_0_32);
tmp_0_0= _dotp2(s_0_03_12,mask_add1_add1_0);    //点积，实质是加法
tmp_1_0= _dotp2(d_0_03_12,mask_add2_add1_0);
tmp_2_0= _dotp2(s_0_03_12,mask_add1_sub1_0);
tmp_3_0= _dotp2(d_0_03_12,mask_add1_sub2_0);
//1 column of tmp
s_1_03_12 = _add2(diff_1_01,diff_1_32);
d_1_03_12 = _sub2(diff_1_01,diff_1_32);
tmp_0_1= _dotp2(s_1_03_12,mask_add1_add1_0);
tmp_1_1= _dotp2(d_1_03_12,mask_add2_add1_0);
tmp_2_1= _dotp2(s_1_03_12,mask_add1_sub1_0);
tmp_3_1= _dotp2(d_1_03_12,mask_add1_sub2_0);
//2 column of tmp
s_2_03_12 = _add2(diff_2_01,diff_2_32);
d_2_03_12 = _sub2(diff_2_01,diff_2_32);
tmp_0_2= _dotp2(s_2_03_12,mask_add1_add1_0);
tmp_1_2= _dotp2(d_2_03_12,mask_add2_add1_0);
tmp_2_2= _dotp2(s_2_03_12,mask_add1_sub1_0);
tmp_3_2= _dotp2(d_2_03_12,mask_add1_sub2_0);
//3 column of tmp
s_3_03_12 = _add2(diff_3_01,diff_3_32);
d_3_03_12 = _sub2(diff_3_01,diff_3_32);
tmp_0_3= _dotp2(s_3_03_12,mask_add1_add1_0);
tmp_1_3= _dotp2(d_3_03_12,mask_add2_add1_0);
tmp_2_3= _dotp2(s_3_03_12,mask_add1_sub1_0);
tmp_3_3= _dotp2(d_3_03_12,mask_add1_sub2_0);
//combine of tmp
tmp_0_32 = _pack2(tmp_0_3,tmp_0_2);             //低16位打包
tmp_0_01 = _pack2(tmp_0_0,tmp_0_1);
tmp_1_32 = _pack2(tmp_1_3,tmp_1_2);
tmp_1_01 = _pack2(tmp_1_0,tmp_1_1);
tmp_2_32 = _pack2(tmp_2_3,tmp_2_2);
tmp_2_01 = _pack2(tmp_2_0,tmp_2_1);
tmp_3_32 = _pack2(tmp_3_3,tmp_3_2);
tmp_3_01 = _pack2(tmp_3_0,tmp_3_1);
//}
//{
//0 column of dct
s_0_03_12 = _add2(tmp_0_01,tmp_0_32);
d_0_03_12 = _sub2(tmp_0_01,tmp_0_32);
dct_0_0 = _dotp2(s_0_03_12,mask_add1_add1_1);
dct_1_0 = _dotp2(d_0_03_12,mask_add2_add1_1);
dct_2_0 = _dotp2(s_0_03_12,mask_add1_sub1_1);
dct_3_0 = _dotp2(d_0_03_12,mask_add1_sub2_1);
//1 column of dct
s_1_03_12 = _add2(tmp_1_01,tmp_1_32);
```

```
        d_1_03_12 = _sub2(tmp_1_01,tmp_1_32);
        dct_0_1 = _dotp2(s_1_03_12,mask_add1_add1_1);
        dct_1_1 = _dotp2(d_1_03_12,mask_add2_add1_1);
        dct_2_1 = _dotp2(s_1_03_12,mask_add1_sub1_1);
        dct_3_1 = _dotp2(d_1_03_12,mask_add1_sub2_1);
        //2 column of dct
        s_2_03_12 = _add2(tmp_2_01,tmp_2_32);
        d_2_03_12 = _sub2(tmp_2_01,tmp_2_32);
        dct_0_2 = _dotp2(s_2_03_12,mask_add1_add1_1);
        dct_1_2 = _dotp2(d_2_03_12,mask_add2_add1_1);
        dct_2_2 = _dotp2(s_2_03_12,mask_add1_sub1_1);
        dct_3_2 = _dotp2(d_2_03_12,mask_add1_sub2_1);
        //3 column of dct
        s_3_03_12 = _add2(tmp_3_01,tmp_3_32);
        d_3_03_12 = _sub2(tmp_3_01,tmp_3_32);
        dct_0_3 = _dotp2(s_3_03_12,mask_add1_add1_1);
        dct_1_3 = _dotp2(d_3_03_12,mask_add2_add1_1);
        dct_2_3 = _dotp2(s_3_03_12,mask_add1_sub1_1);
        dct_3_3 = _dotp2(d_3_03_12,mask_add1_sub2_1);
        dct_0_32 = _pack2(dct_0_3,dct_0_2);
        dct_0_10 = _pack2(dct_0_1,dct_0_0);
        dct_1_32 = _pack2(dct_1_3,dct_1_2);
        dct_1_10 = _pack2(dct_1_1,dct_1_0);
        dct_2_32 = _pack2(dct_2_3,dct_2_2);
        dct_2_10 = _pack2(dct_2_1,dct_2_0);
        dct_3_32 = _pack2(dct_3_3,dct_3_2);
        dct_3_10 = _pack2(dct_3_1,dct_3_0);
        _amemd8(&dct[ 0]) = _itod(dct_0_32,dct_0_10);        //双32位打包并对齐存储
        _amemd8(&dct[ 4]) = _itod(dct_1_32,dct_1_10);
        _amemd8(&dct[ 8]) = _itod(dct_2_32,dct_2_10);
        _amemd8(&dct[12]) = _itod(dct_3_32,dct_3_10);
    }
  }
  static void sub16x16_dct( int16_t dct[16][4][4], uint8_t *pix1, int i_pix1, uint8_t
*pix2, int i_pix2 )
  {
   sub8x8_dct( &dct[ 0], pix1, i_pix1, pix2, i_pix2 );
   sub8x8_dct( &dct[ 4], &pix1[8], i_pix1, &pix2[8], i_pix2 );
   sub8x8_dct( &dct[ 8], &pix1[8*i_pix1], i_pix1, &pix2[8*i_pix2], i_pix2 );
   sub8x8_dct( &dct[12], &pix1[8*i_pix1+8], i_pix1, &pix2[8*i_pix2+8], i_pix2 );
  }
```

上述代码的优化思路是直接处理 8×8 块，将 16 位数据打包成双 16 位，然后利用打包指令提高运算效率。加、减、混合加减使用了点积指令实现。调用 4 次的 8×8DCT 实现 16×16 块的 DCT。

在优化中，使用寄存器代替临时数组；数据源的读取和结果的存储使用地址对齐指令；

为提高处理效率，使用数据打包；强制循环的迭代次数。这些技术也是优化其他核心模块的重要方法，读者可以根据 DCT 的优化过程优化其他模块。

15.3 运行X264编码系统

光盘 chapter15\x264-080907 提供了作者发稿时下载的最新 X264 程序。现在使用 VC++ 2005 编译该程序，首先实现对 YUV420 视频文件编码，然后使用 JM 参考软件验证 H.264 码流的正确性和图像效果，最后使用大家普遍使用的开源 H.264 解码器 ffmpeg，解码 X264 编码的 H.264 码流。

15.3.1 VC平台下编译和运行X264

X264 工程默认使用的是 VC++ 6.0 编译平台，在使用 VC++ 2005 编译该工程时会出现许多编译错误。另外 X264 项目是控制台应用程序，其命令参数多而复杂，下面详细地列出了命令参数及其意义。

1. 编译 X264

若使用 VC++ 2005 编译最新的 X264 工程，则会出现几个编译报警或错误提示。

error LNK2019: 无法解析的外部符号_strtok_r，该符号在函数_parse_zones 中被引用。

这是因为 strtok_r 函数来自于 UNIX 平台，带_r 和不带_r 的函数的区别的是，带_r 的函数是线程安全的，r 的意思是 reentrant，可重入的。解决办法是把所有的函数调用修改为 strtok。

为了便于实时观察编码进度，修改 x264.c 文件中函数 Encode()的以下代码。

```
#if 0
        /* 间隔1000帧更新显示 */
        if( opt->b_progress && param->i_log_level < X264_LOG_DEBUG &&
            ( i_frame_total ? i_frame * 1000 / i_frame_total > i_progress
                            : i_frame % 10 == 0 ) )
#else
      if (1)  //每编码一帧就更新显示
#endif
```

上述代码是对编码信息的显示控制。使用 VC++ 2005 编译 x264-060308 版本时，出现如下提示信息。

```
mc.asm:102: warning: label alone on a line without a colon might be in error
```

这是因为.asm 文件中的标号后需要有冒号“:”。

```
warning C4996: “getenv”被声明为否决的
```

这是因为添加预处理器定义 _CRT_SECURE_NO_DEPRECATE。

2. 命令参数及意义

X264 应用程序是控制台应用程序，通过命令窗口配置编码器的编码参数。x264.c 中列出了编码器的参数内容及其意义，如下所示。

- 使用格式：X264 默认选项 -o 输出文件、输入文件 [长×宽]。
- 输入支持格式：RAW/y4m/avi/avs（编译时可选）。
- 输出支持格式：264/mkv/mp4（编译时可选）。

X264 的许多参数可以有-/--两种输入法。以下等价参数用“参数 1/参数 2 <必需数值格式>”表示：-h/--help 帮助。

命令参数根据作用可分为帧类型选项（表 15-2）、码率控制选项（表 15-3）、模式抉择分析选项（表 15-4）、视频标准化选项（表 15-5）、输入/输出选项（表 15-6）。下面对这些选项以表格形式给出，选项包括名称、值和描述。

表 15-2　帧类型选项

选项名称	值	描述
-I/--keyint	整数	最大 IDR 帧间距，默认为 250
-i/--min-keyint	整数	最小 IDR 帧间距，默认为 25
--scenecut	整数	画面动态变化阈值，当超出此值时插入 I 帧，默认为 40
-b/--bframes	整数	在 IP 帧之间可插入的 B 帧数目最大值，范围 0～16，默认迷为 0
--no-b-adapt		关闭自适应 B 帧判定（–b 设为 1 时可用，其他不推荐）
--b-bias	整数	控制插入 B 帧判定，范围–100～+100，越高越容易插入 B 帧，默认为 0
--b-pyramid		允许 B 帧做参考帧
--no-cabac		关闭内容自适应二进制算术编码（CABAC，高效率的熵编码）（会提高速度，但严重影响质量）
-r/--ref	整数	最大参考帧数，范围 0～16，默认为 1
--nf		关闭环路滤波
-f/--filter	alpha:beta	设置环路滤波的 AlphaC 和 Beta 的参数，范围–6～6，默认都为 0

表 15-3　码率控制选项

选项名称	值	描述
-q/--qp	整数	固定量化模式并设置使用的量化值，范围 0～51，0 为无损压缩，默认为 26
-B/--bitrate	整数	设置平均码率
--crf	整数	质量模式，量化值动态可变
--qpmin	整数	设置最小量化值，范围 0～51，默认为 10
--qpmax	整数	设置最大量化值，范围 0～51，默认为 51
--qpstep	整数	设置相邻帧之间的量化值差，范围 0～50，默认为 4
--ratetol	小数	平均码率模式下，瞬时码率可以偏离的倍数，范围 0.1～100.0，默认为 1.0
--vbv-maxrate	整数	平均码率模式下，最大瞬时码率，默认为 0（与–B 设置相同）

（续表）

选项名称	值	描述
--vbv-bufsize	整数	码率控制缓冲区的大小，单位 Kbit，默认为 0
--vbv-init	小数	码率控制缓冲区数据保留的最大数据量与缓冲区大小之比，范围 0～1.0，默认为 0.9
--ipratio	小数	I 帧和 P 帧之间的量化系数，默认为 1.40
--pbratio	小数	P 帧和 B 帧之间的量化系数
--chroma-qp-offset	整数	色度和亮度之间的量化差，范围–12～+12，默认为 0
-p/--pass	1 或 2 或 3	多次压缩码率控制 1：第一次压缩，创建统计文件 2：按建立的统计文件压缩并输出，不覆盖统计文件 3：按建立的统计文件压缩，优化统计文件
--stats	字符串	统计文件的名称，默认"x264_2pass.log"
--rceq	字符串	速率控制公式，默认"blurCplx^（1-qComp）"
--qcomp	小数	线性量化控制，0.0 为固定码率，1.0 为固定量化值，默认 0.6，只用于 2-pass 和质量模式
--cplxblur	小数	根据相邻帧平滑量化值比例的最大值，范围 0～99.9，默认 20.0，只用于 2-pass 和质量模式
--qblur	小数	对统计文件结果平滑量化值比例的最大值，范围 0～99.9，默认 0.5，只用于 2-pass
--zones	<z0>/<z1>/…	分段量化，格式为：<开始帧>、<结束帧>、<选项> 可选项为：q=<整数>或 b=<小数>

表 15-4　模式抉择分析选项

选项名称	值	描述
-A/--analyse	字符串	块划分方法，默认“p8x8,b8x8,i8x8,i4x4”，可选项：p8x8/p4x4/b8x8/i8x8/i4x4；none/all
--direct	字符串	动态预测方式，默认"spatial"。可选项：none/spatial/temporal/auto
-w/--weightb		允许 B 帧加权预测
--me	字符串	对全像素块动态预测搜索的方式，默认"hex"，可选项如下。 dia：菱形搜索，半径 1； hex：正六边形搜索，半径 2； umh：可变半径六边形搜索； esa：全局搜索
--merange	整数	选项--me 为 umh/esa 时的搜索半径，最大 64，默认 16
-m/--subme	整数	动态预测和分区方式，可选项 1～7，默认 5 1：用全像素块进行动态搜索，对每个块再用快速模式进行四分之一像素块精确搜索 2：用半像素块进行动态搜索，对每个块再用快速模式进行四分之一像素块精确搜索 3：用半像素块进行动态搜索，对每个块再用质量模式进行四分之一像素块精确搜索 4：用快速模式进行四分之一像素块精确搜索 5：用质量模式进行四分之一像素块精确搜索 6：进行 I、P 帧像素块的速率失真最优化（rdo） 7：进行 I、P 帧运动矢量及块内部的速率失真最优化

（续表）

选项名称	值	描述
--b-rdo		B 帧也进行 rdo，需要--subme 在 6 以上
--mixed-refs		可以在一帧内使用不同参考帧
--no-chroma-me		不进行色度的动态预测
--bime		可以平均 B 帧参考块的运动矢量
-8/--8x8dct		可以使用 8×8 的 DCT
-t/--trellis	整数	Trellis 量化，对每个 8×8 的块寻找合适的量化值，需要 CABAC，默认 0。 0：关闭； 1：只在最后编码时使用； 2：一直使用
--cqm	字符串	设置外部量化矩阵格式，默认"flat"，可选项：jvt/flat
--cqmfile	字符串	读取 JM 格式的外部量化矩阵文件，自动忽略其他--cqm*选项
--cqm4	list	设置 4x4 的量化矩阵，用逗号分开，范围 1～255 的 16 个整数
--cqm8	list	设置 8x8 的量化矩阵，用逗号分开，范围 1～255 的 64 个整数
--cqm4i/--cqm4p/ --cqm8i/--cqm8p		设置 I、P 帧不同的量化矩阵
--cqm4iy/--cqm4ic/ --cqm4py/--cqm4pc		设置亮度、色度不同的量化矩阵

表 15-5　视频标准化选项

选项名称	值	描述
--sar width:height		设置长宽比
--overscan	字符串	过扫描线，默认"undef"，可选项：show/crop
--videoforma	字符串	视频格式，默认"undef"，可选项：component/pal/ntsc/secam/mac/undef
--fullrange	字符串	Specify full range samples setting，默认"off"，可选项：off/on
--colorprim	字符串	原始色度格式，默认"undef"，可选项： undef/bt709/bt470m/bt470bg，smpte170m/smpte240m/film
--transfer	字符串	转换方式，默认"undef"，可选项： undef/bt709/bt470m/bt470bg/linear,log100/log316/ smpte170m/smpte240m
--colormatrix	字符串	色度矩阵设置，默认"undef"，可选项： undef/bt709/fcc/bt470bg,smpte170m/smpte240m/ GBR/YcgCo
--chromaloc	整数	色度样本指定，范围 0～5，默认 0

表 15-6 输入/输出选项

选项名称	值	描述
--level	字符串	设定等级
--fps	小数	设定帧率
--seek	整数	设定起始帧
--frames	整数	最大编码帧数
-o/--output		指定输出文件
--threads	整数	编码线程
--thread-input		在编码线程中运行 Avisynth
--no-asm		关闭全部 CPU 优化指令
--no-psnr		关闭 PSNR 计算
--quiet		安静模式
-v/--verbose		显示每一个帧的信息
--progress		显示编码进程
--visualize		显示运动矢量
--sps-id	整数	设置 SPS 和 PPS 的 ID 值，默认 0
--aud		使用数据单元定义符号

在使用 X264 编码器时，默认参数为-B 200 -o foreman_enc.264 352x288.cif。

- -B 表示码流大小，单位为 Kbps。
- -o 表示编码后的码流存储路径及文件名。

后者是命名规则为“图像宽度×图像高度.cif”的 YUV420 文件路径及名称。编码器根据 YUV 的名字分析图像的宽度和高度，编码所有视频帧直到文件结尾。

15.3.2 JM验证X264码流

H.264 协议的 JM 参考软件提供了编码和解码工程，JM 实行性极低，一般用来验证、剖析用户 H.264 码流的兼容性。X264 工程没有提供解码，所以用户可以使用 JM 来解码 X264 码流的正确性。JM 的解码工程 ldecode 是控制台应用程序，运行工程的命令行参数为-i foreman_enc.264 -o foreman_dec.yuv。

最新版本的 JM 提供了 VC++ 7.0/8.0 编译平台。所以用户可以直接使用 VC++ 2005（即 VC++ 8.0）编译 JM 软件。光盘\chapter15\JM14.2 提供了 JM 的最新源代码，双击该目录下的 jm_vc8.sln 打开项目。按“F7”键编译生成解决方案，在 VC++ 2005 的命令行输入以下参数。

```
-i e:\\bitavc\\foreman_enc.264 -o e:\\bitavc\\foreman_dec.yuv
```

按“F5”键执行程序，出现如图 15-14 所示的效果图，图中列出了当前解码器的有关信息。

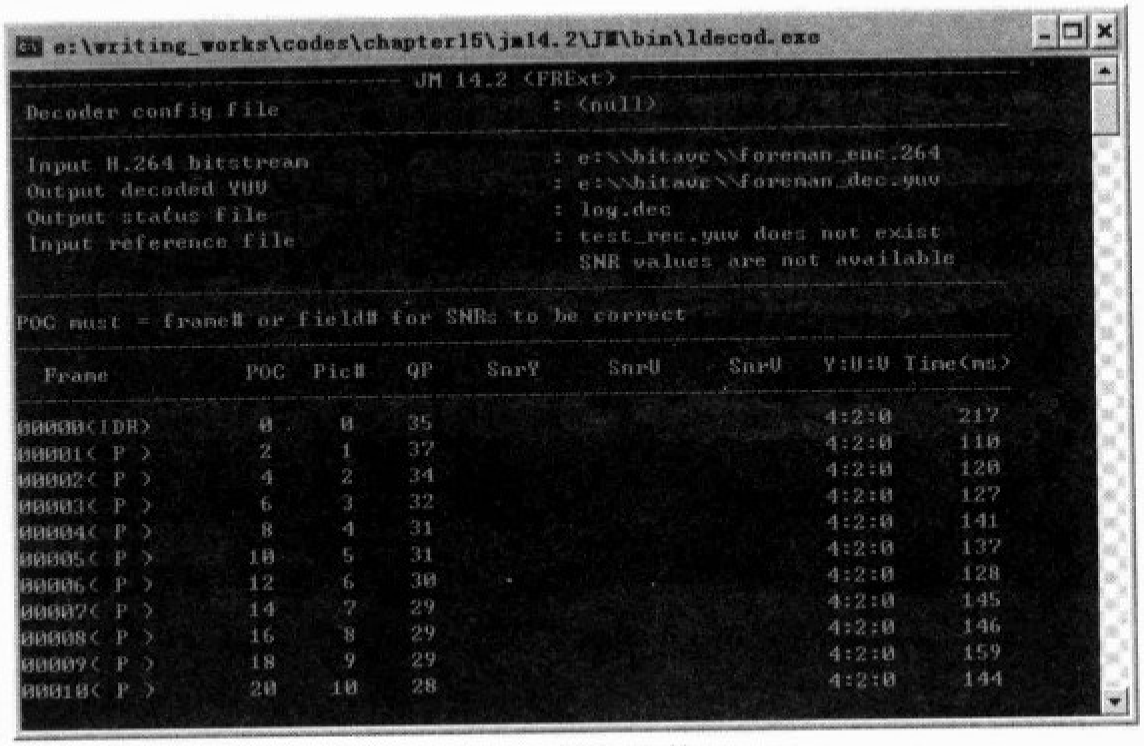


图 15-14 JM 解码软件 ldecode

15.3.3 ffmpeg中的H.264视频解码

ffmpeg 是开源的优秀的多媒体解码器集合，支持常见流媒体格式的解码。由于是 Linux 下的工程，无法在 Windows 下调试和编译。若使用其 H.264 解码算法，则需要从中提取有关的程序代码，然后在 VC 下建立工程。

光盘 chapter15\ffmpeg_h264_vc 提供了从 ffmpeg 提取的 VC 平台下的 H.264 解码工程源代码，用户在解码时可根据需要修改 H.264.c 中的如下代码。

```
inpf = fopen("e:\\bitavc\\test.264", "rb"); /*修改为要解码的H.264文件路径及名称*/
outf = fopen("e:\\bitavc\\out.yuv", "wb");  /*修改为解码输出的YUV文件路径及名称*/
```

15.4 系统效果展示

X264 实现了 H.264 的视频编码，但没有提供解码算法。开源的优秀多媒体解码集合 ffmpeg 提供了 H.264 的解码算法。现在分别使用 X264 对原始 YUV 视频文件做编码，使用 ffmpeg 实现 X264 码流的解码。

15.4.1 X264实现H.264/AVC视频编码

H.264/AVC 编码的码流相比 H.263/MPEG-4 更低、图像质量有大幅提升。现使用 VC++ 2005 编译最新 X264 代码 x264-080907，编码器命令行参数为 -B 300 -o e:\\bitavc\\foreman_enc.264 e:\\yuv\\352x288.cif，其他参数使用编码器默认参数。X264 编码过程中的编码信息如图 15-15 所示，X264 编码结束的统计信息如图 15-16 所示。

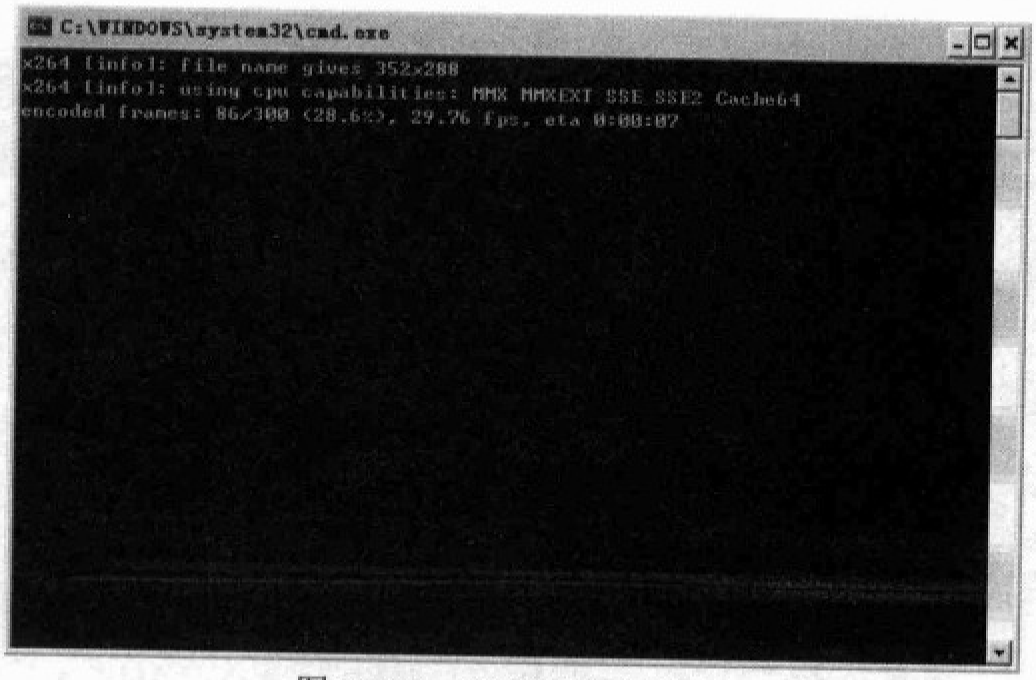


图 15-15　X264 运行过程中

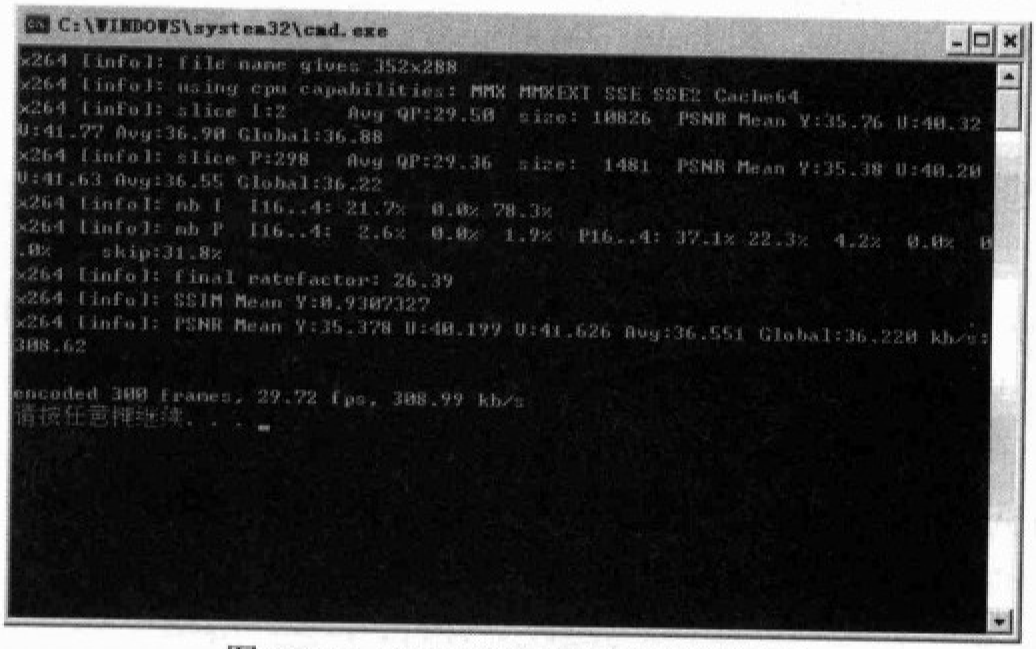


图 15-16　X264 运行后信息统计结果

编码器在结束时给出了编码的详细统计信息：I/P 帧的宏块的类型、图像质量 PSNR、编码器编码速度、实际码流大小等。

15.4.2　ffmpeg-h264-vc工程实现H.264视频解码

网络上流行着多个 H.264 视频解码器，但绝大部分不提供源代码。ffmpeg 提供了 H.264 的解码源代码，作硬件解码、移植 DSP 的也基本选择 ffmpeg 为蓝本。解码上述 H.264 文件结果如图 15-17 所示。

如图 15-18 所示是使用 YUVviewer.exe 程序观看到的图像效果对比。码流大小是 300Kbps，Baseline 档级：I/P 帧、CAVLC 编码和 1 个参考帧等。从图 15-18 可以看出，解码后图像的主观质量与原始图像差别不大，图像信息损失少、码流低、图像逼真。

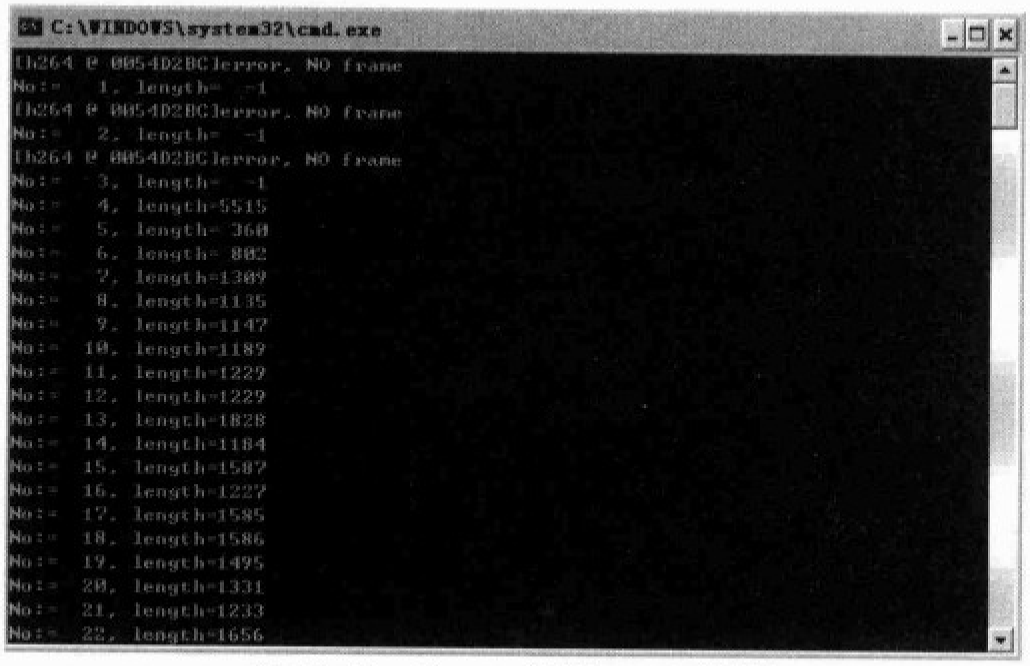
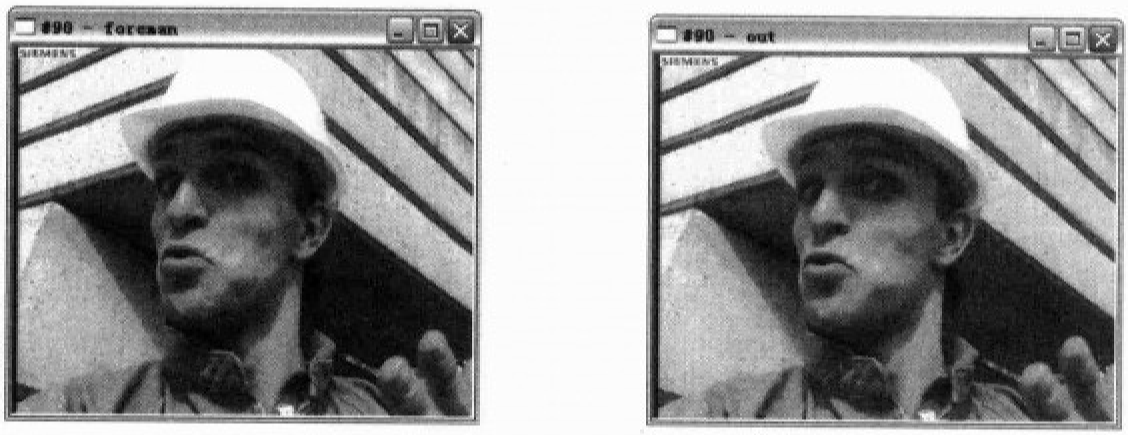


图 15-17 ffmpeg 解码 H.264 视频

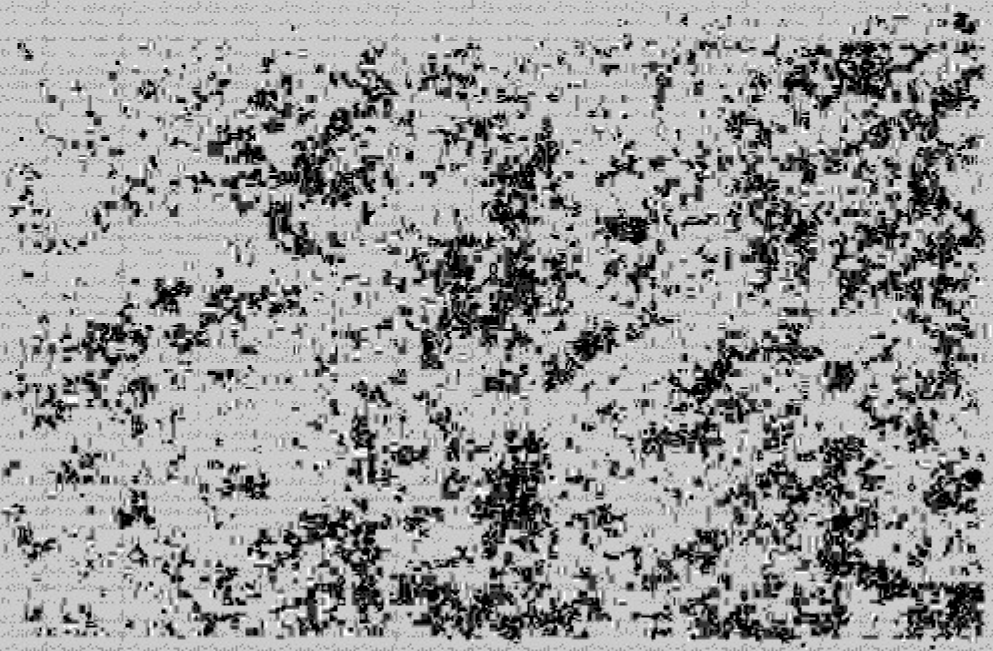


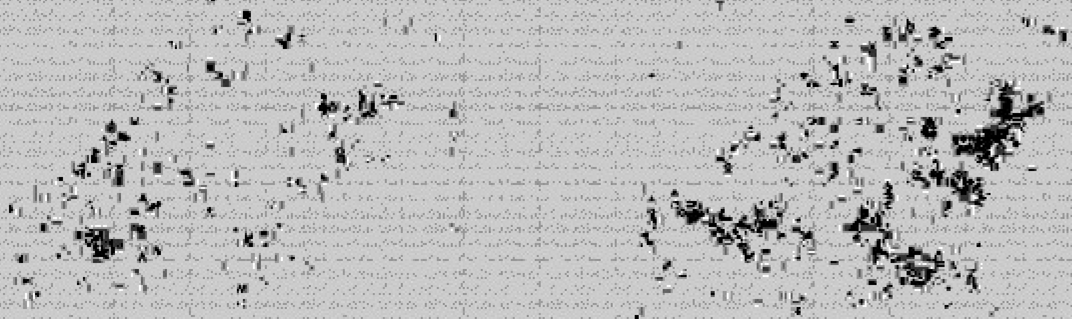
（a）原始图像 （b）编/解码后图像

图 15-18 X264 编/解码图像效果对比

15.5 本章小结

本章介绍了 H.264/AVC 视频编/解码算法的新特性，分析了 X264 实现 H.264/AVC 算法的技术要点，对编码过程作了阐述，特别对 Inter 块的模式抉择作了详细的剖析。接着对 X264 底层函数的汇编优化作了分析，同时给出了 DM642 优化核心模块的一般步骤。使用 VC++ 2005 编译运行 X264，使用 JM、ffmpeg 开源软件解码 X264 的 H.264 码流，最后给出了 X264 系统的运行效果。





第 16 章

打造自己的 DVR 监控系统

DVR（Digital Video Recoder）数字视频记录是数字硬盘录像机和视频监控系统的核心。数字视频监控 DVR 应用已经从最初的金融领域发展到了交通、公安、环境、智能建筑、广场、机房、考场等各种场所，只要是想利用图像和声音进行监控的场所都可以使用 DVR 系统。DVR 已经成为视频监控领域一个炙手可热的产品，并且 DVR 已经向智能化、网络化发展，系统智能理解、分析图像给出报警联动信息，使监控中心作出响应：全面记录当前状态、启动预定报警机制。本章将讲解 PCI 式 DVR 的有关开发技术，详细介绍设计友好监控主界面的技术过程。

本章重点

- DVR 监控系统分析与设计
- DVR 监控系统主界面软件设计
- 模拟 DVR 视频采集
- 系统效果展示

16.1 DVR监控系统分析与设计

DVR 的监控中心往往都是设计界面友好、功能齐全且便于操作。在前面章节介绍了视频算法核心及其实现平台 X86、DSP 设计开发的基础上，本章借助通用类设计一个友好主界面，包括全屏显示背景图片、位图按钮、多画面显示等设计。实现视频监控系统的基本功能有多画面切换显示、图像参数控制、通道选择及其他功能控制。为了模拟 DVR 的视频采集、回放功能，案例使用 DirectShow SDK 实现视频的采集、预览显示和多影音文件回放。用户可以直接使用或者基于此开发出自己的监控系统中心软件。

监控系统的组成大同小异：多通道的音/视频采集、编码、传输、存盘、定时记录、预定报警、图像参数控制、多通道的音/视频检索、回放等。硬件压缩仍然是视频监控的主流，具有体积小、功耗小、环境适应性强和设备成本低等特点。硬件压缩 DVR 制造商一般都给用户提供了 SDK 软件开发包，从而便于用户如系统集成商开发出自己的监控系统应用产品。本书以 PCI 式 DVR 为对象，讲解如何设计一个友好、漂亮的监控主界面。

16.1.1 监控系统分类

DVR 监控系统种类繁多，技术实现指标多样化，根据系统结构主要有 PCI 式 DVR 和嵌入式 DVR。这些绝大部分是硬件实现编码。PCI 式插卡类可以使用软件编码，即 PCI 图像采集卡采集图像，软件编程实现对采集视频的编码压缩。

1. PCI 式 DVR 系统特点

以传统的 PC 为硬件基础，以 Windows 操作系统（或 Linux）为基本软件，配备 PCI 视频采集或采集压缩卡，编制软件成为一套完整的系统。PC 是一种通用的平台，PC 的硬件更新换代速度快，因而 PCI 式 DVR 的产品性能提升较容易，同时软件修正、升级也比较方便。PCI 式 DVR 还可细分为工控机 DVR 和商用机 DVR，前者适用于恶劣环境、价格高，后者价格便宜、对环境的适应性好。

PCI 式 DVR 产品的主要问题如下。

- PC DVR 产品操作及维护需要有一定的技术基础，加大系统维护难度。
- PC DVR 的数据存储及操作系统均在硬盘中，文件安全性降低，受操作系统的工作状态影响。
- PC 各类配件的发展非常快，且均为专业厂家生产，DVR 系统维护难度大。

PCI 式 DVR 产品的优点如下。

- 存储空间较大（容易扩展存储硬盘），适宜长时间录像。
- 良好的人机接口和文件管理等。
- 软、硬件升级比较容易，产品更新快。

- 维修成本比较低，一般的故障可通过更换部件进行维修，整机不会报废。

应该说市面上主流的DVR系统是PCI式的，一方面终端用户Windows操作系统使用普遍，另一方面DVR系统支持功能一般要强于嵌入式DVR。

2. 嵌入式DVR系统特点

嵌入式DVR就是基于嵌入式处理器和嵌入式实时操作系统的嵌入式系统，它采用专用芯片或可编程芯片对图像进行压缩及解码回放，嵌入式操作系统主要完成整机的控制及管理。此类产品没有PC DVR那么多的模块和多余的软件功能，在设计制造时对软、硬件的稳定性进行了针对性的规划，因此此类产品运行更稳定，不会有死机的问题产生，而且在音/视频压缩码流的存储速度、分辨率及画质上都有较大的改善，就功能来说丝毫不比PCI式DVR逊色。

嵌入式DVR的现状是拥有自主开发能力的公司极少，其技术门槛和资金门槛都比较高，一般小型公司都无低层嵌入开发能力，通常只能采用国外一些低档的方案，受原方案限制，导致灵活性不够，无法满足客户多样化需求，有的无网络功能，有的有网络功能但不强。嵌入式系统的软、硬件设计难度通常都很高，但是它为用户提供了以下好处。

- 易于使用，无须具有PC操作技能；嵌入式DVR的操作一般通过面板按键或遥控器进行，只要会使用DVD/VCD等家电就会使用嵌入式DVR。
- 系统稳定性高，软件容错能力更强，无须专人管理。嵌入式DVR采用嵌入式实时多任务操作系统，视频监视、压缩、存储、网络传输等功能集中到一个体积较小的设备内，系统的实时性、稳定性、可靠性大大提高，所以无须专人管理，适合于无人值守的环境；PC DVR的软件一般都安装在硬盘上，系统的异常关机都可能造成系统文件被破坏或者系统硬盘被损坏，从而导致整个系统崩溃，可靠性很差。
- 软硬件成本低，嵌入式DVR采用一体化的硬件结构，无须像PC DVR那样要购买显卡、内存等设备，硬件成本比PC DVR低。软件采用专用的嵌入式操作系统，用户不需要支付该部分的费用，而PC DVR如果采用正版的Windows系统则其费用将在2000元以上，如果使用盗版操作系统，还可能面临微软公司的侵犯知识产权的诉讼。
- 机械尺寸较小，结构简单紧凑。嵌入式DVR的硬件采用一体化设计，不同于PCI式DVR插卡式的结构；整个系统结构简单，体积小，重量轻，同时也提高了系统的稳定性和可靠性。

本书以PCI式DVR视频监控系统为讲解对象，设计基于Windows平台的视频监控系统的主界面。

16.1.2 监控系统组成

监控系统由于应用领域广泛，不同的应用需求对应监控系统有不同的配置，如网络传输与本地存储、有无音频支持、是否支持系统联动报警等。监控系统基本的功能有音/视频采集、编码压缩、数据存储、数据网络传输、系统联动报警等。PCI式的DVR视频监控把采集的视频经过视频压缩卡编码压缩，经PCI总线传输到本机，本机数据存储或通过网络传输到局域

网中供其他终端观看。嵌入式 DVR 可以是一个视频服务器，客户端通过网络访问 DVR 的编码数据，或者 DVR 带有本地存储功能。另外视频监控系统一般都有云台控制功能，监控中心根据画面内容系统自动或人为干预控制摄像头跟踪感兴趣的运动物体，这种具有云台控制的监控系统更多的是嵌入式 DVR。典型的视频监控系统的拓扑结构如图 16-1 所示。

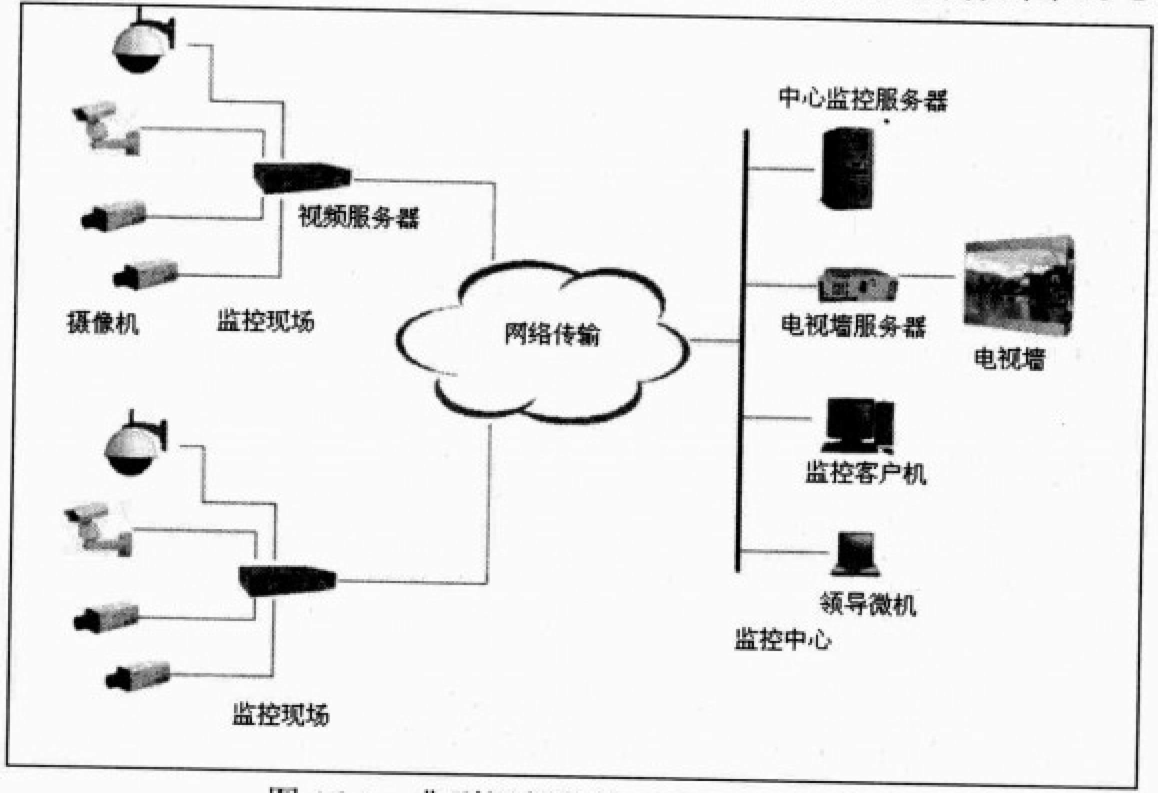


图 16-1　典型视频监控系统拓扑结构图

监控系统主要分为监控现场、网络传输和监控中心 3 部分。监控现场可以有多个，每个现场有多个数据采集设备，视频服务器完成对采集数据的编码压缩和数据的维护等工作，同时根据监控中心的命令控制摄像头等采集设备完成特定动作。网络传输可以是局域网、广域网 Internet、无线网络或 PSTN 公用电话网。监控中心根据功能不同有中心监控服务器、解码显示电视墙服务器、监控普通客户分机和领导微机。中心监控服务器配有监控中心软件，负责对监控前端的视频服务器的维护、压缩视频流的存储、解码显示、图像分析、系统联动报警等。电视墙服务器是解码监控现场的压缩码流，送高分辨率显示设备显示观看。普通的监控客户机也可以在不同地方在授权许可下登录视频服务器。领导微机是权限特殊的客户分机。

- 监控现场除了视频采集设备外，还应配有语音、门禁传感器等其他信号采集设备，视频服务器对多种监控信号监控、分析，提供重要信息给监控中心。监控中心根据既定规则做出相应处理。
- 视频服务器是完成对数据的编码压缩、图像智能处理和数据服务管理功能，可以是 PCI 式的 DVR 或嵌入式的 DVR。因为 DVR 能够记录一定时间长度的音/视频数据，根据其功能又被称为“数字硬盘录像机”。视频服务器或数字硬盘录像机的核心就是 DVR 编码压缩记录，它是监控系统的核心技术，是体现系统最主要数据指标的部件。
- 监控中心是主界面友好的客户应用层，实时解码预览、存储记录各个通道的图像并能任意切换通道，控制图像参数，控制摄像头或云台等采集设备，设置跟踪方式：定点、定时轮循监控跟踪，定时录像、录像参数配置等。

监控系统是一个庞大的系统工程，涉及视频编/解码、通信协议的开发、网络传输、系统集成应用。所以监控系统是由多个公司、部门共同研发出来的，主要分两部分：一是视频压

缩卡或视频服务器端等系统底层，二是系统集成等应用层。第一部分是专门做视频编/解码、数字硬盘录像机的多媒体板卡制造商，他们提供硬件和 SDK 开发包，第二部分是使用多媒体板卡等压缩设备的系统集成开发者，他们开发监控中心软件和与普通用户直接交互的程序，是基于 SDK 的二次开发。

16.1.3 基于压缩板卡的SDK软件开发包

PCI 式的 DVR 核心部件是 PCI 压缩板卡，它完成海量视频数据的采集和编码压缩等核心任务。压缩板卡制造商一般都提供了使用板卡进行二次开发的 SDK 软件包，以及基于 SDK 的 DEMO 测试程序。用户可以不用关心板卡的细节和底层驱动，而基于 SDK 可以快速编制出有自己特色的监控中心软件。本书使用的是德加拉的压缩板卡 AVE 6800。

1. 压缩板卡

AVE 6800 音/视频编码压缩卡是德加拉公司为当今安防市场开发的一款高性价比音/视频实时同步压缩卡，视频码流调节范围宽，视频压缩码率模式可支持 CBR/VBR，板卡尺寸小，功耗低，4 路卡的功耗<6W。CPU 资源占用低，板卡可支持 64 路以上同时压缩，每路均可独立控制启停和调节参数，互不干扰。图像清晰，无论是预览还是回放的图像，都很透亮，没有锯齿和毛边。功能齐全，板卡通过了常温和高温长时间的测试，可靠性高且稳定性好。压缩板卡实物如图 16-2 所示。

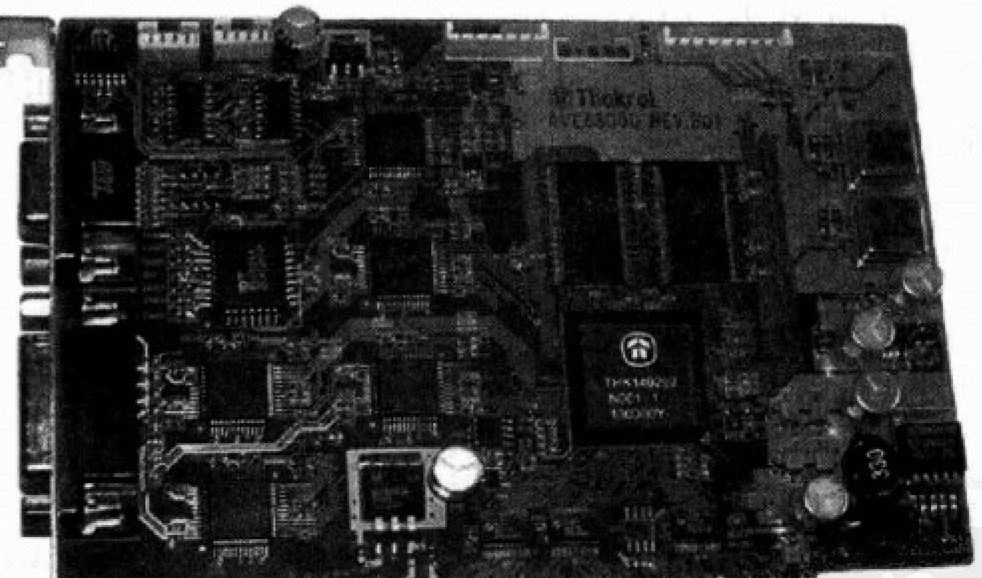


图 16-2 PCI 式多媒体压缩板卡

AVE 6800 系列压缩卡提供完整的 SDK 开发包，本地 SDK、播放 SDK、网传 SDK 为标准通用架构，德加拉所有板卡完全兼容，主要技术特点如下。

- H.264 硬件预览、硬件压缩，音/视频实时同步。
- H.264 音/视频实时压缩和复合通过硬件完成，不占用 CPU 资源。
- 视频压缩码率模式可支持 CBR/VBR，并易于控制。
- 视频压缩码率调节范围宽（16Kbps～1Mbps）。
- OSD 采用软件实现效果和稳定性好，支持遮挡，徽标设置。

- 可实现 CIF 图像压缩。
- 板卡小，功耗低，稳定性高。
- 压缩帧速率可调。
- 支持水印防伪技术。
- 音频监听接口。
- 可支持 64 路以上同时压缩，每路均可独立控制启停和调节参数，互不干扰。
- 灵敏的多窗口动态检测，不占用 CPU 资源。
- 在压缩过程中，可软件控制音频监听接口。
- 报警输入输出。
- 适用于图像在低带宽网络上传输。

2. SDK 软件开发包

SDK 软件开发包多数是以动态链接库 DLL 的形式发布的。系统应用开发能够使用多种编程语言，如 VB、VC++、Delphi、C++ Builder 等开发软件，调用板卡的 DLL。

基于 SDK 的二次开发，首先需要购买压缩板卡硬件，然后安装板卡驱动，最后根据 SDK 软件包的说明和 demo 测试程序，再根据用户的需求，编制监控中心软件。

我们可以从德加拉公司的的官方网站下载压缩包：http://www.thakral-it.com/up_files/download/20080808150520PM-9227.rar。

压缩包中的文档 AVE6900SDK_Local_Manual_Chn.doc 即为 SDK 的中文版技术文档。如图 16-3 所示是 SDK 的函数格式说明。

定　义：

```
BOOL   TKLSDK_OpenChannel(
    IN      int         nChannelNo,
    IN  OUT int         *pChannelType,
    OUT     long        *pHandle
    );
```

参　数：

【IN】nChannelNo　　：卡号，从 1 开始计数。

【IN OUT】pChannelType ：卡类型，取值范围，AVEDEFAULT,AVE1000,AVE2000,AVE3000,AVE4000,AVE5000,AVE6000,AVE7000,AVE8000,AVE9000。

【OUT】pHandle　　：返回通道句柄指针。

返回值：

正确返回 true，错误返回 false。

功　能：

当执行成功时，函数返回一个代表卡的句柄并修改 pChannelType 所指内容为实际卡型。应用程序将保存句柄供后续使用。否则返回 NULL，可以用来判断该路卡没有打开或者该路卡是否存在。

说　明：

对德加拉系列卡进行操作之前都必须确保已经调用本函数成功地打开了欲操作的通道。

1000	2000	4000	5000	6000	7000	7200	6800	
√	√	√	√	√	√	√	√	

图 16-3　SDK 函数格式说明

开发者根据 SDK 的函数格式约定，详细地、灵活地掌握 SDK 功能及其使用方法，编制监控系统的中心软件，设计漂亮、友好的主程序。

16.2 DVR监控系统主界面软件设计

考虑到监控系统的在线监控人员是普通人员，DVR 监控系统基本都是设计或傻瓜式、交互友好的应用程序，主界面漂亮而又简单，按钮和界面类似于 Windows XP 界面，功能非常强大，考虑全面，包括硬盘空间统计、当前系统时间显示、定时录像等设计人性化的功能。本案例以 VC++ 2005 为开发软件，使用通用的按钮皮肤类 CxSkinButton 实现监控系统主界面的设计。

16.2.1 对话框应用程序开发

监控中心软件使用基于对话框的 MFC 编程，建立对话框应用程序框架，基本的创建流程如下。

STEP 01 新建项目，命名项目为 VSS（Video Surveillance System，意为视频监控系统）。如图 16-4 所示。

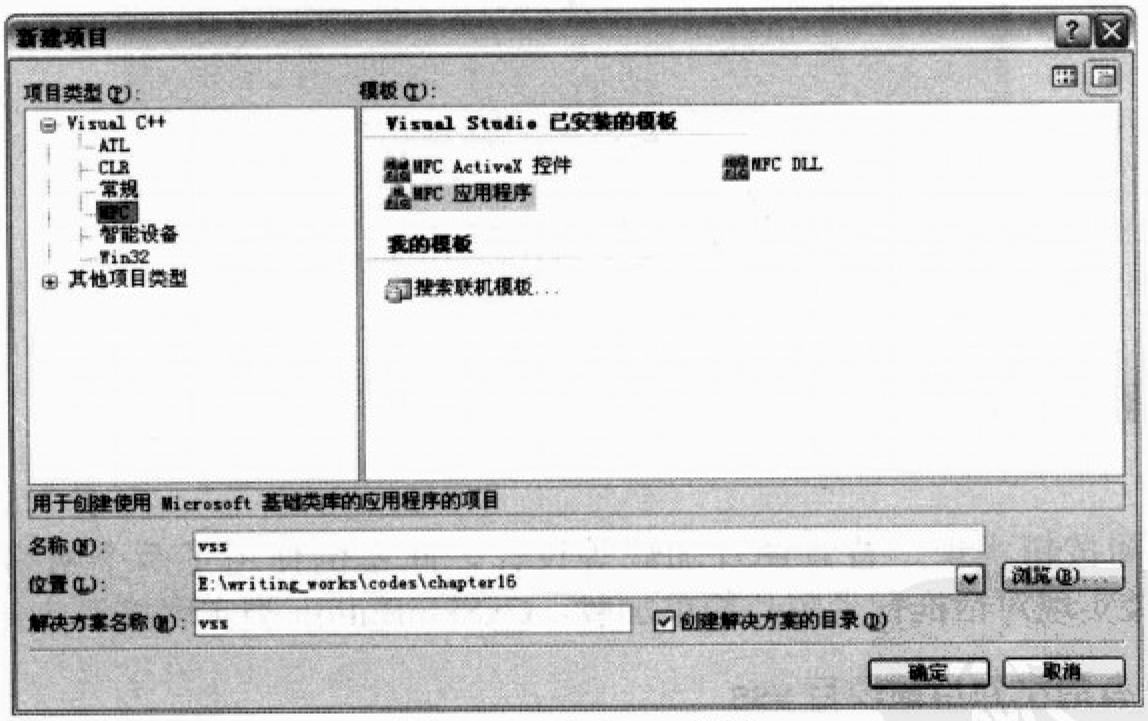


图 16-4　建立 vss 项目

STEP 02 选择“基于对话框”的应用程序类型，确认选中“使用 Unicode 库”复选框，如图 16-5 所示。选择“应用程序向导”的默认选项，单击“下一步”按扭直到完成向导。然后按“F7”键生成解决方案。按“F5”键运行程序，如图 16-6 所示。

STEP 03 如果在编译过程中遇到“没有找到 mfc80d.dll”等编译错误提示信息，则请参考附录中的 VC++ 2005 开发常见问题。

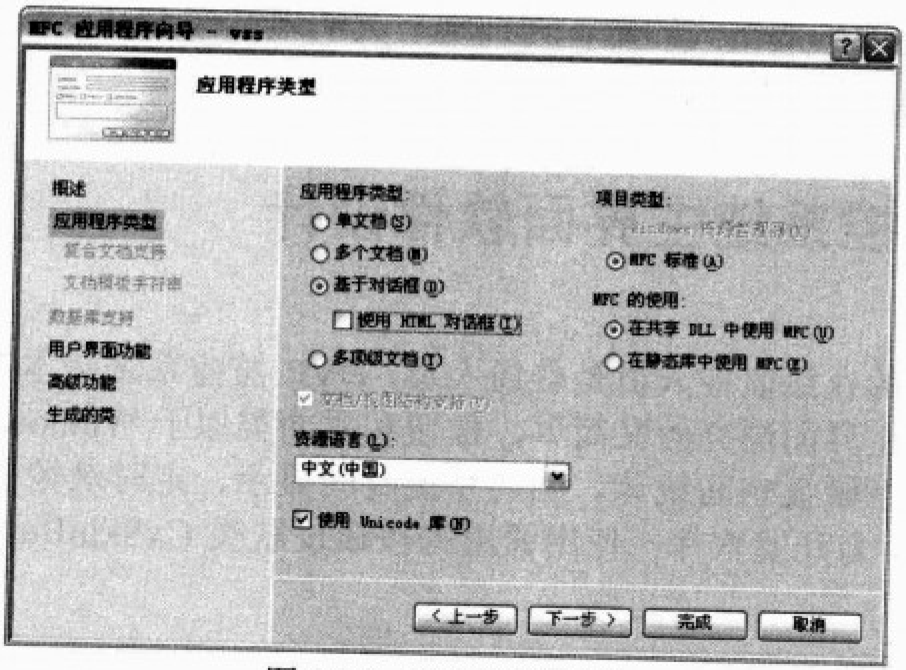


图 16-5　应用程序类型

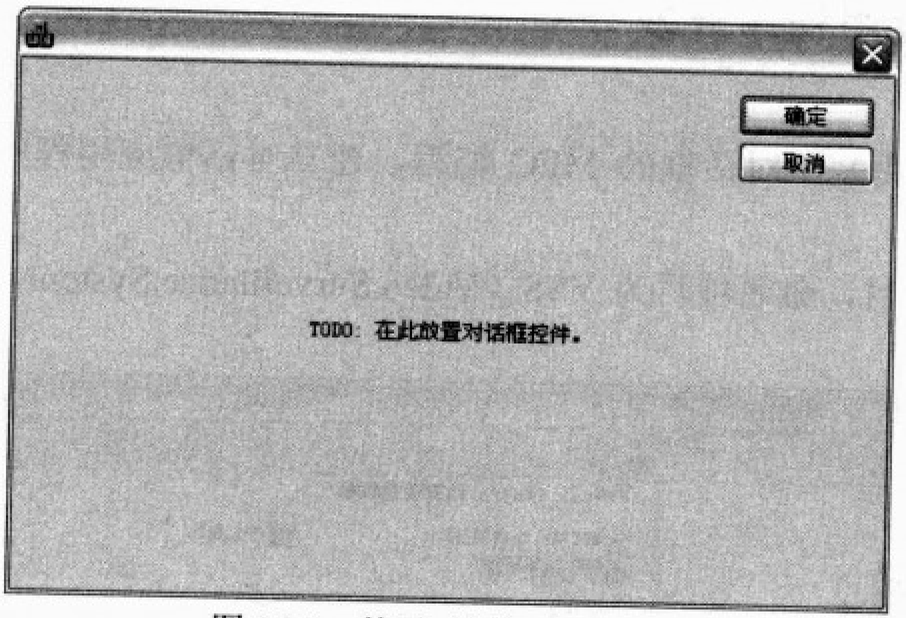


图 16-6　基于对话框应用程序

16.2.2　位图按钮设计

为完成位图按钮实现、背景图片加载等设计，此案例使用了两个网络上公开的类：CBkDialogST 类实现对话框背景图片简单加载，CxSkinButton 类实现按钮变化位图的加载。

1．添加已有的类到当前项目 vss

首先把 CbkDialogST 和 CxSkinButton 的头文件和实现文件分别复制到当前项目目录下。然后向项目中“添加/添加现有项…”，这些文件分别是 BkDialogST.cpp、BkDialogST.h、xSkinButton.cpp、xSkinButton.h。

2．对话框资源和类

选中“资源视图”中的“Dialog”子项，单击鼠标右键，在弹出的快捷菜单中选择“添加资源”→“Dialog”命令，然后单击“新建”按扭。修改该对话框的 ID 为“IDD_DIALOG_MAINFACE”，将这个对话框作为监控系统主界面。

在 ID 为 IDD_DIALOG_MAINFACE 的对话框内部单击鼠标右键，在弹出的快捷菜单中选择“添加类”命令。命名类为 CmainFrame，见图 16-7MFC 类向导。

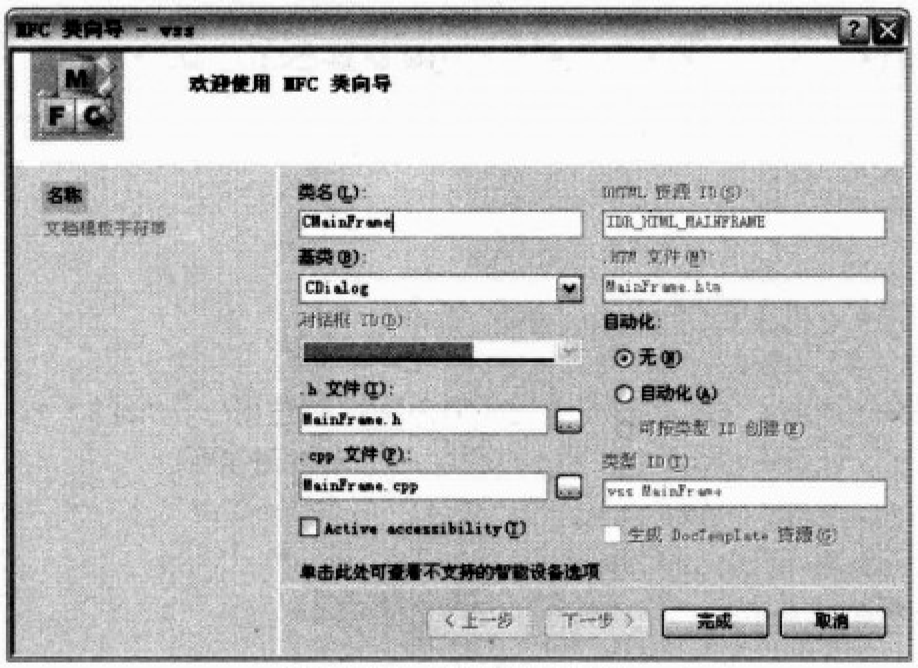


图 16-7　MFC 类向导

单击“完成”按扭后，向导自动生成类的实现文件和头文件 MainFrame.cpp 和 MainFrame.h。

3. 修改对话框的基类

在 CMainFrame 类生成时，VC 默认基类是 CDialog。为了使类 CBkDialogST 成为 CMainFrame 的基类，首先在包 MainFrame.h 中含头文件：

```
#pragma once
#include "bkdialogst.h"
#include "xskinbutton.h"
//CMainFrame 对话框
```

然后手工修改 CMainFrame 类的基类 CDialog 为 CBkDialogST，包括实现和头文件。

4. 使用 CMainFrame 对话框

在文件 vss.cpp 中使用 CMainFrame 对话框，同时关闭 CvssDlg 对话框。

```
//vss.cpp：定义应用程序的类行为。
#include "stdafx.h"
#include "vss.h"
#include "vssDlg.h"
#include "MainFrame.h"        //主界面对话框类
#ifdef _DEBUG
#define new DEBUG_NEW
#endif
//CvssApp
BEGIN_MESSAGE_MAP(CvssApp, CWinApp)
 ON_COMMAND(ID_HELP, &CWinApp::OnHelp)
END_MESSAGE_MAP()
```

```
//CvssApp构造
CvssApp::CvssApp()
{
//TODO: 在此处添加构造代码，将所有重要的初始化放置在InitInstance 中
}
//唯一的一个CvssApp对象
CvssApp theApp;
//CvssApp初始化
BOOL CvssApp::InitInstance()
{
//如果一个运行在Windows XP 上的应用程序清单指定要使用ComCtl32.dll版本6或更高版本来启用
可视化方式，则需要InitCommonControlsEx()。否则，将无法创建窗口
INITCOMMONCONTROLSEX InitCtrls;
InitCtrls.dwSize = sizeof(InitCtrls);
//将它设置为包括所有要在应用程序中使用的公共控件类。
InitCtrls.dwICC = ICC_WIN95_CLASSES;
InitCommonControlsEx(&InitCtrls);
CWinApp::InitInstance();
AfxEnableControlContainer();
//标准初始化
//如果未使用这些功能并希望减小，最终可执行文件的大小，则应移除下列不需要的特定初始化例程，更
改用于存储设置的注册表项
// TODO: 应适当修改该字符串，例如修改为公司或组织名
SetRegistryKey(_T("应用程序向导生成的本地应用程序"));
////////////////////////////////////////////////////////////
CMainFrame  dlgmain;           //定义主对话框界面对象
dlgmain.DoModal();             //显示主界面
////////////////////////////////////////////////////////////
#if 0 //屏蔽下面的代码
CvssDlg dlg;
m_pMainWnd = &dlg;
INT_PTR nResponse = dlg.DoModal();
if (nResponse == IDOK)
{
// TODO: 在此处放置处理何时用“确定”按钮来关闭对话框的代码
}
else if (nResponse == IDCANCEL)
{
//TODO: 在此处放置处理何时用“取消”按钮来关闭对话框的代码
}
#endif
//由于对话框已关闭，所以将返回FALSE以便退出应用程序，而不是启动应用程序的消息泵
return FALSE;
}
```

监控系统的主界面没有直接使用 VC++ 2005 应用程序向导生成的对话框，而是使用重新

添加的 CMainFrame 对话框。首先添加主界面对话类的 MainFrame.h 头文件，然后在创建实例函数 InitInstance()中调用主界面对话框类，最后使用条件编译指令#if 把向导生成的对话框类屏蔽。

5. 向 CMainFrame 主对话框类添加按钮

首先添加 8 个按钮 CButton 类，按钮的 ID 和功能描述如表 16-1 所示。

表 16-1 功能按钮

按钮 ID	捆绑变量	功能描述
IDC_BUTTON_ONE_PIC	m_BtnOnePic	单画面显示
IDC_BUTTON_FOURE_PIC	m_BtnFourePic	四画面显示
IDC_BUTTON_EXIT_APP	m_BtnExitApp	退出应用程序
IDC_BUTTON_CHAN_L	m_BtnChan1	第一通道
IDC_BUTTON_CHAN_M	m_BtnChan2	第二通道
IDC_BUTTON_CHAN_R	m_BtnChan3	第三通道
IDC_BUTTON_MNI	m_BtnMini	最小化
IDC_BUTTON_CLOSE	m_BtnClose	关闭

然后对这些按钮捆绑控件变量，方法：用鼠标右键单击“添加变量”按扭，启动“添加成员变量向导”对话框，如图 16-8 所示。选中“控件变量”复选框。分别对这 8 个按钮添加、捆绑控件变量，如表 16-1 所示。

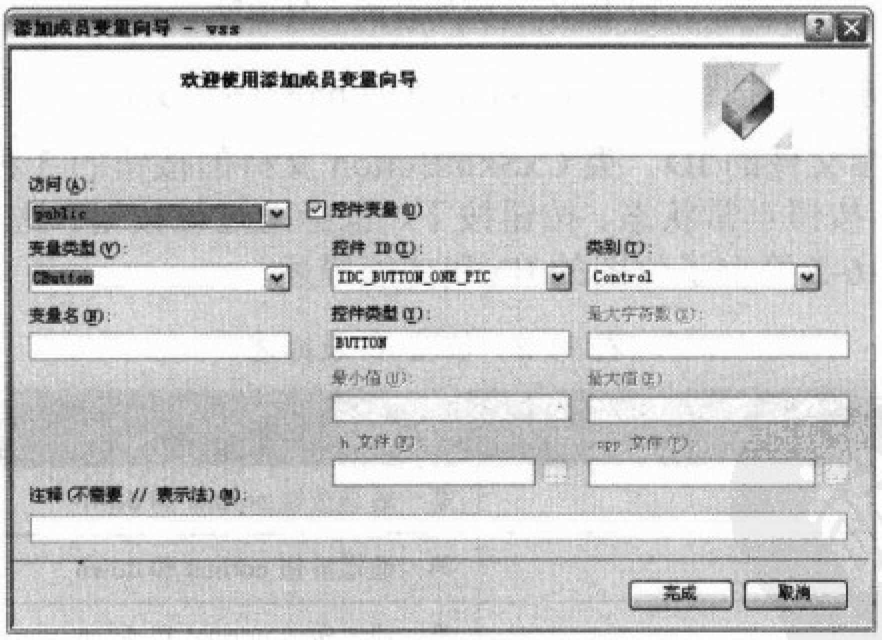


图 16-8 添加成员变量向导

最后，为了使这些按钮的基类派生于 CxSkinButton，需要手工修改。如 VC 自动生成的控件变量如下。

```
public:
 CButton m_BtnOnePic;
public:
 CButton m_BtnFourePic;
```

```
//............
```

现手工修改上述代码为：

```
public:
 CxSkinButton m_BtnOnePic;
public:
 CxSkinButton m_BtnFourePic;
//..............
```

这样，按钮的基类就是案例中要使用的 CxSkinButton 类，使用其加载位图函数 SetSkin 完成位图按钮的加载。

6. 位图资源

为了使按钮实现位图加载，首先添加位图资源到项目中，过程如下。

STEP 01 在“视图/资源视图”中单击鼠标右键，在弹出和快捷菜单中选择“添加资源”命令，弹出如图 16-9 所示的对话框。

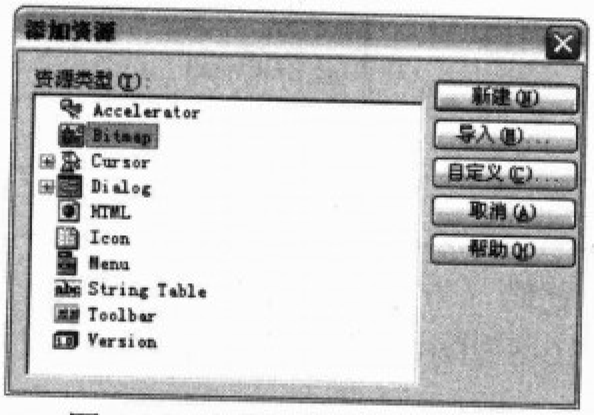


图 16-9 “添加资源”对话框

STEP 02 单击“导入”按钮，把 res 目录下的.bmp 图片文件导入到程序中。

STEP 03 修改位图文件的 ID。类 CxSkinButton 支持的按钮的 3 种状态分别为 normal、down、over，分别表示按钮正常状态、按钮按下、鼠标滑过该按钮。这里没有使用 over 状态，只使用了前两种。表 16-2 给出了位图的 ID 和功能描述。

表 16-2 位图 ID 及描述

位图 ID	功能描述
IDB_1_d/ IDB_1_n	第一通道按钮 normal 和 down
IDB_2_d/ IDB_2_n	第二通道按钮 normal 和 down
IDB_3_d/ IDB_3_n	第二通道按钮 normal 和 down
IDB_BTN_D/ IDB_BTN_N	按钮 normal 和 down
IDB_CLOSE	关闭系统
IDB_EXIT	退出系统
IDB_4_PIC	四画面显示
IDB_MINI	最小化系统
IDB_1_PIC	单画面显示

7. 虚拟函数 OnInitDialog

在 VC 自动生成的对话框类中，自动包含对话框初始化函数 OnInitDialog。而在自己建立的对话框类中，则没有定义该函数。但是有些初始化工作需要在 OnInitDialog 中实现，可以手工添加该虚拟函数，步骤如下。

STEP 01 选中类 CmainFrame 并单击鼠标右键，在弹出的快捷菜单中选择“属性”命令，出现“重写”提示。

STEP 02 选中“OnInitDialog”右端下拉箭头：“<添加> OnInitDialog”，重写以增加对话框初始化。

STEP 03 初始化对话框：

```
ModifyStyle(WS_CAPTION,0,0);              //删除标题栏
this->ShowWindow(SW_SHOWMAXIMIZED);       //窗口最大化
SetBitmap( IDB_1_PIC );                   //设置背景图片
```

STEP 04 生成项目解决方案，然后运行，如图 16-10 所示。

图 16-10　主界面初期设计

8. 实现位图按钮

实现位图按钮主要分两步，第一步移动按钮到合适位置，第二步加载位图。

STEP 01 定义按钮的 CWnd 类对象。

在 CMainFrame 类中定义各个按钮的 CWnd 对象指针。

```
public:
 CMainFrame(CWnd* pParent = NULL);   //标准构造函数
 virtual~CMainFrame();
 CWnd *pBtnOnePic,*pBtnFourePic;
 CWnd *pBtnExitApp;
 CWnd *pBtnChan1,*pBtnChan2,*pBtnChan3;
 CWnd *pBtnMini,*pBtnClose;
```

定义的按钮变量的基类是底层的基类 CWnd，使用这些指针直接控制按钮动作和属性。

STEP 02 定义接口初始化内部函数 InitInterface()。在该函数中，完成按钮类的指针对象获取、按钮指定位置移动、加载按钮位图、添加 tooltip 提示。

```
void CMainFrame::InitInterface()
{
 //获取对象指针
 pBtnOnePic   = GetDlgItem(IDC_BUTTON_ONE_PIC);
 pBtnFourePic = GetDlgItem(IDC_BUTTON_FOURE_PIC);
 pBtnExitApp = GetDlgItem(IDC_BUTTON_EXIT_APP);
 pBtnMini    = GetDlgItem(IDC_BUTTON_MINI);
 pBtnClose   = GetDlgItem(IDC_BUTTON_CLOSE);
 pBtnChan1   = GetDlgItem(IDC_BUTTON_CHAN_L);
 pBtnChan2   = GetDlgItem(IDC_BUTTON_CHAN_M);
 pBtnChan3   = GetDlgItem(IDC_BUTTON_CHAN_R);
 //移动到指定位置
 pBtnOnePic->MoveWindow(16,702,37,37);
 pBtnFourePic->MoveWindow(16+75,702,37,37);
 pBtnMini->MoveWindow(880,2,35,35);
 pBtnClose->MoveWindow(965,3,35,35);
 pBtnChan1->MoveWindow( 876,401,26,17);
 pBtnChan2->MoveWindow( 876+26, 402,25,16);
 pBtnChan3->MoveWindow( 876+51, 402,26,16);
 pBtnExitApp->MoveWindow(920,684,46,48);
 //加载位图
 m_BtnOnePic.SetSkin(IDB_BTN_N,IDB_BTN_D,IDB_BTN_N,IDB_BTN_N,0,0,0,0,0);
 m_BtnFourePic.SetSkin(IDB_BTN_N,IDB_BTN_D,IDB_BTN_N,IDB_BTN_N,0,0,0,0,0);
 m_BtnMini.SetSkin(IDB_MINI,IDB_MINI,IDB_MINI,IDB_MINI,0,0,0,0,0);
 m_BtnClose.SetSkin(IDB_CLOSE,IDB_CLOSE,IDB_CLOSE,IDB_CLOSE,0,0,0,0,0);
 m_BtnChan1.SetSkin(IDB_1_n,IDB_1_d,IDB_1_n,IDB_1_n,0,0,0,0,0);
 m_BtnChan2.SetSkin(IDB_2_n,IDB_2_d,IDB_2_n,IDB_2_n,0,0,0,0,0);
 m_BtnChan3.SetSkin(IDB_3_n,IDB_3_d,IDB_3_n,IDB_3_n,0,0,0,0,0);
 m_BtnExitApp.SetSkin( IDB_EXIT,IDB_EXIT, IDB_EXIT, IDB_EXIT,0,0,0,0,0);
 //添加在线提示 tooltip
 m_BtnOnePic.SetToolTipText(_T("单画面显示"));
 m_BtnFourePic.SetToolTipText(_T("四画面显示"));
 m_BtnMini.SetToolTipText(_T("最小化"));
 m_BtnClose.SetToolTipText(_T("关闭"));
 m_BtnChan1.SetToolTipText(_T("第1通道"));
 m_BtnChan2.SetToolTipText(_T("第2通道"));
 m_BtnChan3.SetToolTipText(_T("第3通道"));
}
```

在上述代码中，首先获取控件的 CWnd 指针以便于访问和控制，然后使用该指针把按钮等控件移动到指定位置，接着是使用按钮的基类 CxSkinButton 的成员函数 SetSkin 完成不同状态的位图加载，最后使用其成员函数 SetToolTipText 为按钮添加在线提示。

至此，主界面的位图按钮就实现了，见图 16-11 监控主界面。下面开始对按钮的事件作出响应和处理。

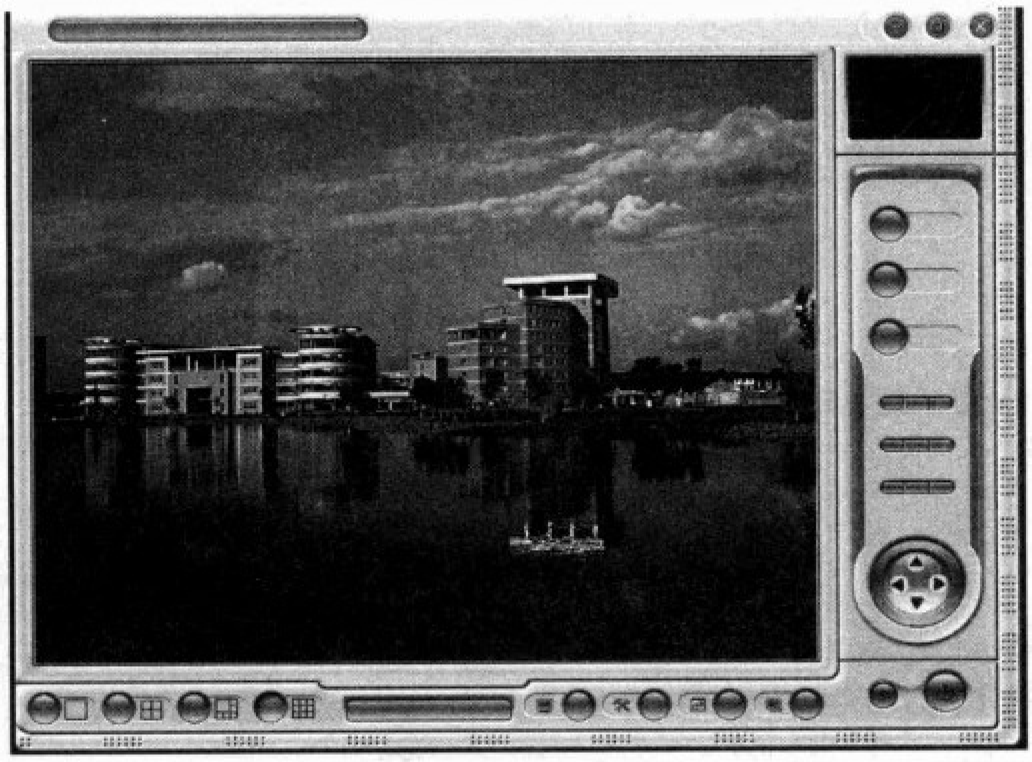


图 16-11 监控主界面

9. 按钮响应函数

这里的响应函数是用鼠标单击按钮时，程序需要作出对应响应的事件处理。利用类 CMainFrame 的基类 CBkDialogST 的成员函数 SetBitmap 来实现背景图片的更新。

STEP 01 单画面显示。

```
void CMainFrame::OnBnClickedButtonOnePic(){
 //TODO：在此添加控件通知处理程序代码
      SetBitmap(IDB_1_PIC); //更新背景图片
      //......
}
```

SetBitmap 是基类 CBkDialogST 的成员函数，位图 IDB_1_PIC 是单画面显示时的系统背景图片。完成加载单画面显示的背景图片。

STEP 02 四画面显示。

```
void CMainFrame::OnBnClickedButtonFourePic(){
 //TODO：在此添加控件通知处理程序代码
 SetBitmap(IDB_4_PIC);
      //......
}
```

加载使用四画面显示时的背景图片 IDB_4_PIC。

STEP 03 退出系统。

在退出系统前，系统将让用户确认是否要离开本系统。

```
void CMainFrame::OnBnClickedButtonExitApp(){
 //TODO：在此添加控件通知处理程序代码
```

```
    if (MessageBox(_T("确定要退出该账户吗？"),
                        _T("退出系统"),MB_OKCANCEL|MB_ICONQUESTION)==IDCANCEL)
        return ;
    else {
        //释放其他资源
        SendMessage(WM_SYSCOMMAND, SC_CLOSE, NULL);
        }
}
//关闭系统
void CMainFrame::OnBnClickedButtonClose(){
 // TODO: 在此添加控件通知处理程序代码
 //释放其他资源
 SendMessage(WM_SYSCOMMAND, SC_CLOSE, NULL);
}
//最小化系统
void CMainFrame::OnBnClickedButtonMini(){
 //TODO: 在此添加控件通知处理程序代码
 SendMessage(WM_SYSCOMMAND, SC_MINIMIZE, NULL);
}
```

在退出系统前，提示用户确实要退出本系统，防止误操作。若确实要离开本系统，则向系统发送 SC_CLOSE 命令消息。主界面的右上角是关闭系统，这里是直接退出系统。退出系统这两个地方都可以退出，注意在退出前，确保其他资源已经释放，防止内存泄露！

STEP 04 通道选择处理。

这里给出了通道响应函数，具体功能实现读者可以根据特定需要丰富这些函数。选择某通道对其作监控或跟踪等处理。

```
void CMainFrame::OnBnClickedButtonChanL(){
 //TODO: 在此添加控件通知处理程序代码
 MessageBox(_T("您选择了第一个通道解码显示"));
}
void CMainFrame::OnBnClickedButtonChanM(){
 //TODO: 在此添加控件通知处理程序代码
 MessageBox(_T("您选择了第二个通道解码显示"));
}
void CMainFrame::OnBnClickedButtonChanR(){
 //TODO: 在此添加控件通知处理程序代码
 MessageBox(_T("您选择了第三个通道解码显示"));
}
```

用户单击了通道选择按钮，对其消息进行处理。这里只演示了添加位置，具体功能读者根据需要实现。

10. OnPaint 消息处理函数

对话框的绘制是在函数 OnPaint 中完成的。但是自己创建的对话框最初并没有该函数。如果需要的话，则要求用户自己手工添加。

STEP 01 选中类 CmainFrame 并单击鼠标右键，在弹出的快捷菜单中选择“属性”命令，出现“消息”提示。

STEP 02 找到 WM_PAINT，然后单击下拉箭头：“<添加> OnPaint”，添加对话框的绘制工作。在该函数中添加对字体的控制，在主界面显示标题内容。

```
void CMainFrame::OnPaint()
{
 CPaintDC pDC(this); //device context for painting
 //TODO: 在此处添加消息处理程序代码
 CFont NewFont;
 NewFont.CreateFont(16,                //nHeight字体高度
                    0,                 //nWidth字体宽度
                    0,                 //nEscapement字体显示的角度
                    0,                 //nOrientation字体的角度
                    FW_MEDIUM,         //FW_NORMAL、FW_BOLD、nWeight 字体的磅数
                    FALSE,             //bItalic斜体字体
                    FALSE,             //bUnderline带下画线的字体
                    0,                 //cStrikeOut 带删除线的字体
                    GB2312_CHARSET,    //ANSI_CHARSET、nCharSet 所需的字符集
                    OUT_DEFAULT_PRECIS,    //nOutPrecision输出的精度
                    CLIP_DEFAULT_PRECIS,   //nClipPrecision裁减的精度
                    DEFAULT_QUALITY,       //nQuality逻辑字体与输出设备的实际
                                           //字体之间的精度
                    DEFAULT_PITCH | FF_SWISS,     //nPitchAndFamily字体间距和
                                                   字体集
                    _T("幼圆")                     //字体名称
                    );
CFont *pOldFont = pDC.SelectObject(&NewFont);
pDC.SetBkMode(TRANSPARENT);                //先进设备描述表
pDC.SetTextColor(RGB(255,0,0));            //设置字体颜色，这里是红色
pDC.TextOut(100,10,_T("西南科技大学视频监控系统"));
pDC.SelectObject(&pOldFont);               //回复到旧字体
NewFont.DeleteObject();                    //删除新创建的字体
//不为绘图消息调用CBkDialogST::OnPaint()
}
```

上述代码实现在主界面上显示能控制字体属性的文本。注意在显示完毕后，要释放创建的字体资源，否则会产生内存泄露。

11. 动态显示系统时间

添加、显示系统当前时间的过程如下。

STEP 01 添加静态文本控件 CStatic，ID 为 IDC_STATIC_TIME。

STEP 02 为控件添加指针变量，在 CMainFrame 类中添加如下加黑代码。

```
public:
```

```
CMainFrame(CWnd* pParent = NULL);  //标准构造函数
virtual ~CMainFrame();
CWnd *pBtnOnePic,*pBtnFourePic;
CWnd *pBtnExitApp;
   //......
CWnd *pDisSysTime;                 //显示系统时间
```

STEP 03 获取控件 CStatic 的 CWnd 类型指针，并移动控件位置、设置大小。

```
pDisSysTime = GetDlgItem(IDC_STATIC_TIME);   //获取CWnd指针
pDisSysTime->MoveWindow(850,54,130,80);      //移动文本控件到指定位置，并设置大小
```

STEP 04 为了定时获取、显示系统时间，这里使用定时器。首先定义定时器 0：

```
BOOL CMainFrame::OnInitDialog()
{
 CBkDialogST::OnInitDialog();
 //TODO: 在此添加额外的初始化
 ModifyStyle(WS_CAPTION,0,0);
 this->ShowWindow(SW_SHOWMAXIMIZED);
 InitInterface();
 SetBitmap( IDB_1_PIC );
 SetTimer(0,1000,NULL);
 return TRUE; //return TRUE unless you set the focus to a control
 //异常: OCX属性页应返回FALSE
}
```

这里要实现在主界面上实时更新显示系统的当前日期和时间。使用定时器来实现定时刷新界面内容。

STEP 05 添加定时器服务函数 OnTimer。首先选中类 CmainFrame 并单击鼠标右键，在弹出的快捷菜单中选择“属性”命令，打开“属性”对话框。然后单击“消息”按钮，查找 WM_TIMER 消息，添加定时器超时消息处理函数 OnTimer。

```
void CMainFrame::OnTimer(UINT_PTR nIDEvent)
{
 //TODO: 在此添加消息处理程序代码和/或调用默认值
 if (nIDEvent==0) //定时器ID
 {
       CTime tTime = CTime::GetCurrentTime();        //获取系统时间
CString strTime =tTime.Format("\n  %Y-%m-%d  \n\n  %H:%M:%S");     //数据格式化
       this->pDisSysTime->SetWindowText(strTime);    //显示时间
 }
 CBkDialogST::OnTimer(nIDEvent);
}
```

上述代码实现定时器的服务功能，即读取系统时间，把日期和时间显示在主界面的文本控件上。

STEP 06 在退出应用程序时，确保已销毁定时器。

```
void CMainFrame::OnBnClickedButtonExitApp()
```

```
{
 //TODO：在此添加控件通知处理程序代码
 if (MessageBox(_T("确定要退出该账户吗？"),
                      _T("退出系统"),MB_OKCANCEL|MB_ICONQUESTION)==IDCANCEL)
       return ;
 else {
       KillTimer(0);
       SendMessage(WM_SYSCOMMAND, SC_CLOSE, NULL);
 }
}
```

上述代码实现退出应用程序时销毁定时器。

STEP 07 经过以上多个步骤的界面设计和功能实现，监控系统的主界面如图 16-12 所示。

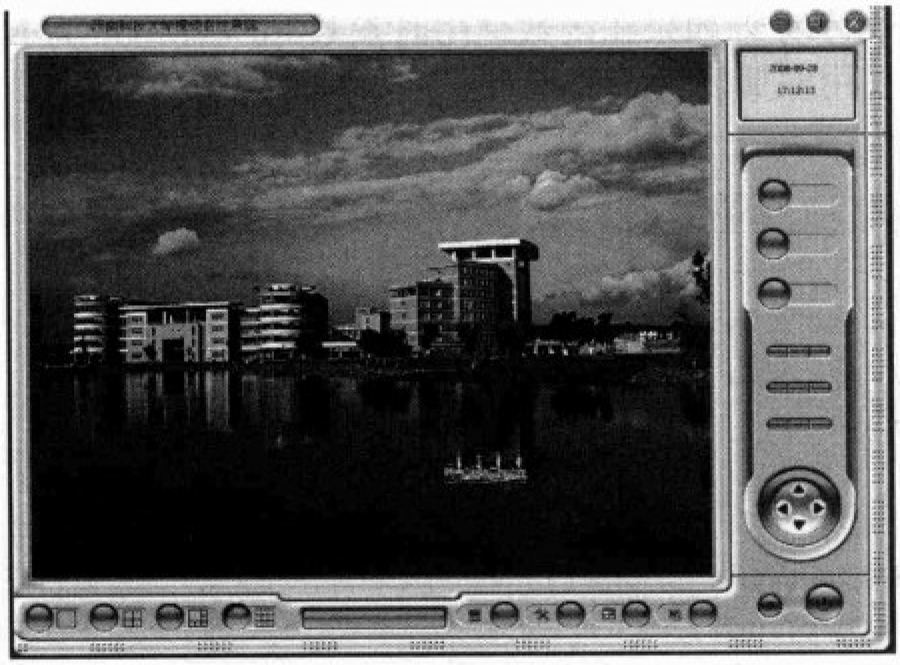


图 16-12　监控系统主界面

12. 动态改变屏幕分辨率

由于此案例的背景图片最大是按照 XGA（1024 像素×768 像素）的分辨率设计的，如果屏幕分辨率高于或低于 XGA，则需要程序修改分辨率，以最佳效果显示背景图片。所以在对话框创建时自动检测屏幕分辨率，然后在退出系统时需要恢复系统的屏幕分辨率。

STEP 01 添加变量记录屏幕最初分辨率，同时记录是否做了调整。

```
int ScrHeight;              //屏幕高度
int ScrWidth;               //屏幕宽度
BOOL bChanged;              //屏幕分辨率是否做了调整
```

STEP 02 检测、设置屏幕分辨率。

首先获取当前屏幕的最大分辨率，如果不是 1024 像素×768 像素，则程序需修改。用到的函数原型为 LONG ChangeDisplaySettings(LPDEVMODE lpDevMode, DWORD dwflags)。

lpDevMode 是一个指向 DEVMODE 数据结构的指针，DEVMODE 的数据结构描述了欲设定显示器的各类属性值。

Dwflags 表明对显示设备的修改方式，具体取值有以下几种。

- 0：动态改变显示设备属性。
- CDS_UPDATEREGISTRY：动态改变显示设备属性并修改注册表相关设置，在下次启动计算机时，本次所作的修改依然有效。
- CDS_TEST：测试所作的修改是否有效。

该函数调的返回值如下。

- DISP_CHANGE_SUCCESSFUL：修改成功。
- DISP_CHANGE_RESTART：修改后需重新启动（在显示器设定中选择了“应用新的颜色前重新启动计算机”）。
- DISP_CHANGE_FAILED：修改失败。
- DISP_CHANGE_BADMODE：修改模式错误（比如你的显示器是单色的，但你却将其修改为256色的）。如果在函数调用时参数lpDevMode为NULL，且dwflags为0，则显示设备使用注册表当前值来设置显示特性。

程序实现过程如下。

```
BOOL CMainFrame::OnInitDialog()
{
 CBkDialogST::OnInitDialog();
///////////////////////////////////////////////////////
 HWND hDesktop;
 HDC hdc;
 DWORD w, h;
 hDesktop = ::GetDesktopWindow();                    //返回桌面窗口的句柄
 hdc = ::GetDC(hDesktop);                            //返回设备环境句柄
 ScrWidth = w = GetDeviceCaps(hdc, HORZRES);         //返回屏幕宽度
 ScrHeight = h = GetDeviceCaps(hdc, VERTRES);        //返回屏幕高度
 bChanged = FALSE;                                   //没有调整标志
 if((w!=1024)&&(h!=768)){
       DEVMODE  lpDevMode;
       lpDevMode.dmBitsPerPel     = 32;              //每像素使用的比特数,32/24/16
       lpDevMode.dmPelsWidth = 1024;                 //水平分辨率
       lpDevMode.dmPelsHeight     = 768;             //垂直分辨率
       lpDevMode.dmDisplayFrequency=60;              //刷新频率，单位Hz
       lpDevMode.dmSize = sizeof(lpDevMode);         //DEVMODE数据结构的大小(单位字节)
            /*在通常情况下，不同的显示设备（如打印机）用到的DEVMODE数据结构的内容不同，比如
设定打印机时，不会用到dmDisplayFrequency属性。所以，在使用DEVMODE数据结构时，应向系统说明用到
的有效数据成员,dmFields的作用即在于此。若在程序中仅用到dmPelsWidth(水平分辨率)和dmPelsHeight
(垂直分辨率)，则该值应为DM_PELSWIDTH|DM_PELSHEIGHT*/
       lpDevMode.dmFields = DM_PELSWIDTH|DM_PELSHEIGHT|DM_BITSPERPEL;
       LONG result = ChangeDisplaySettings(&lpDevMode,0);
       if(result == DISP_CHANGE_SUCCESSFUL) {
             ChangeDisplaySettings(&lpDevMode,CDS_UPDATEREGISTRY);
             //使用CDS_UPDATEREGISTRY表示修改是持久的,并在注册表中写入了相关的数据
             bChanged = TRUE;
```

```
        } else
                ChangeDisplaySettings(NULL,0); //修改失败，用注册表中的当前设置显示
    }
    ////////////////////////////////////////////////////////
    ModifyStyle(WS_CAPTION,0,0);
    this->ShowWindow(SW_SHOWMAXIMIZED);
    InitInterface();
    SetBitmap( IDB_1_PIC );
    SetTimer(0,1000,NULL);
    return TRUE;  // return TRUE unless you set the focus to a control
    //异常: OCX属性页应返回FALSE
}
```

一般在设置屏幕分辨率的时候，屏幕会有短暂的黑屏，这是正常的。不过用户最好在编程前确认显示器是否支持不低于 1024 像素×768 像素分辨率显示，因为更改分辨率可能会对显示器的硬件有影响。

STEP 03 退出应用系统，恢复屏幕分辨率。

在退出应用程序时，系统还应恢复程序运行前的屏幕分辨率。

```
void CMainFrame::OnBnClickedButtonExitApp()
{
  //TODO: 在此添加控件通知处理程序代码
  //释放资源
  if (MessageBox(_T("确定要退出该账户吗? "),
                        _T("退出系统"),MB_OKCANCEL|MB_ICONQUESTION)==IDCANCEL)
        return ;
  else {
        KillTimer(0);
        if (bChanged)
        {
                DEVMODE   lpDevMode;
                lpDevMode.dmBitsPerPel   = 32;
                lpDevMode.dmPelsWidth    = ScrWidth;    //程序检测的原始屏幕水平分辨率
                lpDevMode.dmPelsHeight   = ScrHeight;   //垂直分辨率
                lpDevMode.dmDisplayFrequency=60;
                lpDevMode.dmSize = sizeof(lpDevMode);
                lpDevMode.dmFields = DM_PELSWIDTH|DM_PELSHEIGHT|DM_BITSPERPEL;
                LONG result = ChangeDisplaySettings(&lpDevMode,0);
                if(result == DISP_CHANGE_SUCCESSFUL)
                {
                    ChangeDisplaySettings(&lpDevMode,CDS_UPDATEREGISTRY);
                    bChanged = TRUE;
                }
                else
                {
                    ChangeDisplaySettings(NULL,0);
```

```
            }
        }
        SendMessage(WM_SYSCOMMAND, SC_CLOSE, NULL);
    }
}
```

同样，在退出 XGA 分辨率时，屏幕可能会有几秒的黑屏。为减少这种程序的自动修改，可以在程序运行前自己手工修改。

至此，友好、漂亮的监控主界面已经设计完成，下面通过前面章节介绍的视频采集技术 DirectShow 或 VFW 模拟 DVR 的视频采集，在监控主界面显示采集的图像。

16.3 模拟DVR视频采集

DVR 视频采集是经过压缩板卡完成的，用户如果没有这种视频硬件采集设备，则可以使用常用的摄像头或对已有的影音文件进行回放，在监控主界面模拟 DVR 的视频采集。

16.3.1 基于DirectShow的视频采集回放

在第 9 章介绍了使用 DirectShow 进行视频实时捕捉、显示的案例，这里直接使用该案例技术在 DVR 监控系统中增加视频采集并实时显示的功能。

STEP 01 配置 DirectShow 的开发环境。在 VC++ 2005 的开发环境中配置 DirectShow 的有关头文件和库文件路径，详见第 7 章。

STEP 02 添加视频捕获类。把项目 Capture Video 中的文件 CaptureClass.cpp 和 CaptureClass.h 复制到当前工程目录下。然后添加到 vss 项目中。接着在 MainFrame.h 中修改：包含头文件，并定义捕获类的对象。

```
#include "CaptureClass.h"
//......
 CCaptureClass m_cap;
//......
```

STEP 03 添加图像显示控件。为了在监控主界面实时显示采集的图像，需要添加显示控件。这里使用 VC 2005 工具箱的 Picture Control 控件。

添加 4 个 Picture Control 控件：IDC_VIDEO0、IDC_VIDEO1、IDC_VIDEO2 及 IDC_VIDEO3，并修改这些控件的 Type 属性为 Rectangle。然后定义 4 个 CWnd 指针变量以便于使用这些控件。

```
CWnd *pStaticVideo0,*pStaticVideo1,*pStaticVideo2,*pStaticVideo3;
```

在 InitInterface 中初始化这些指针，并分别移动到合适位置，然后隐藏显示。在对通道选择时可灵活地选择控件的显示属性。

```
    //获取指针
pStaticVideo0= GetDlgItem(IDC_VIDEO0);
```

```
pStaticVideo1= GetDlgItem(IDC_VIDEO1);
pStaticVideo2= GetDlgItem(IDC_VIDEO2);
pStaticVideo3= GetDlgItem(IDC_VIDEO3);
    //移动指定位置，设置合适大小
pStaticVideo0->MoveWindow(26,53,394,311);
pStaticVideo1->MoveWindow(26+395,53,392,311);
pStaticVideo2->MoveWindow(26,53+311+2,394,307);
pStaticVideo3->MoveWindow(26+395,53+311+2,392,307);
    //隐藏显示
pStaticVideo0->ShowWindow(SW_HIDE);
pStaticVideo1->ShowWindow(SW_HIDE);
pStaticVideo2->ShowWindow(SW_HIDE);
pStaticVideo3->ShowWindow(SW_HIDE);
```

上述代码实现显示控件的属性控制，首先获取这些显示控件的 CWnd 类型指针，接着移动到指定位置，设置合适大小以充满主界面的视频显示客户区，最后设置隐藏显示。

STEP 04 修改按钮消息响应函数。用户在选择单画面显示时，确保第一个窗口放大充满显示区域。在选择四画面显示时，把第一个窗口恢复到显示区域的 1/4 大小。

```
void CMainFrame::OnBnClickedButtonOnePic()
{
 //TODO: 在此添加控件通知处理程序代码
 SetBitmap(IDB_1_PIC);
 pStaticVideo0->MoveWindow(26,53,811-24,672-52);  //充满显示区域
}
void CMainFrame::OnBnClickedButtonFourePic()
{
 //TODO: 在此添加控件通知处理程序代码
 SetBitmap(IDB_4_PIC);
 pStaticVideo0->MoveWindow(26,53,394,311);           //恢复到显示区域的1/4
}
```

这里使用第一通道显示采集捕获的视频图像。首先正常显示第一个显示窗口，隐藏其他 3 个显示窗口，然后把第一个显示窗口的窗口句柄传给捕获类实现视频的采集和显示。

```
void CMainFrame::OnBnClickedButtonChanL()
{
 //TODO: 在此添加控件通知处理程序代码
 pStaticVideo0->ShowWindow(SW_SHOWNORMAL);    //正常显示
 pStaticVideo1->ShowWindow(SW_HIDE);          //隐藏
 pStaticVideo2->ShowWindow(SW_HIDE);
 pStaticVideo3->ShowWindow(SW_HIDE);
 HWND hVW = pStaticVideo0->GetSafeHwnd();
 m_cap.PreviewImages(0 , hVW);                //0表示采集设备的索引号
                                              //这里假定系统安装了一个视频采集设备

MessageBox(_T("您选择了第一个通道解码显示"));
}
```

上述代码实现第一通道的消息处理，首先使能显示该窗口区，把其他窗口隐藏。最后获取显示窗口的窗口句柄，调用视频捕获、显示函数捕获的图像。

经过以上程序设计，运行结果如图 16-13 所示。在运行程序时，确保选择四画面显示，然后单击右边的通道按钮“第 1 通道”，如图中圈定的按钮，则程序会自动采集图像并在第一个窗口显示。

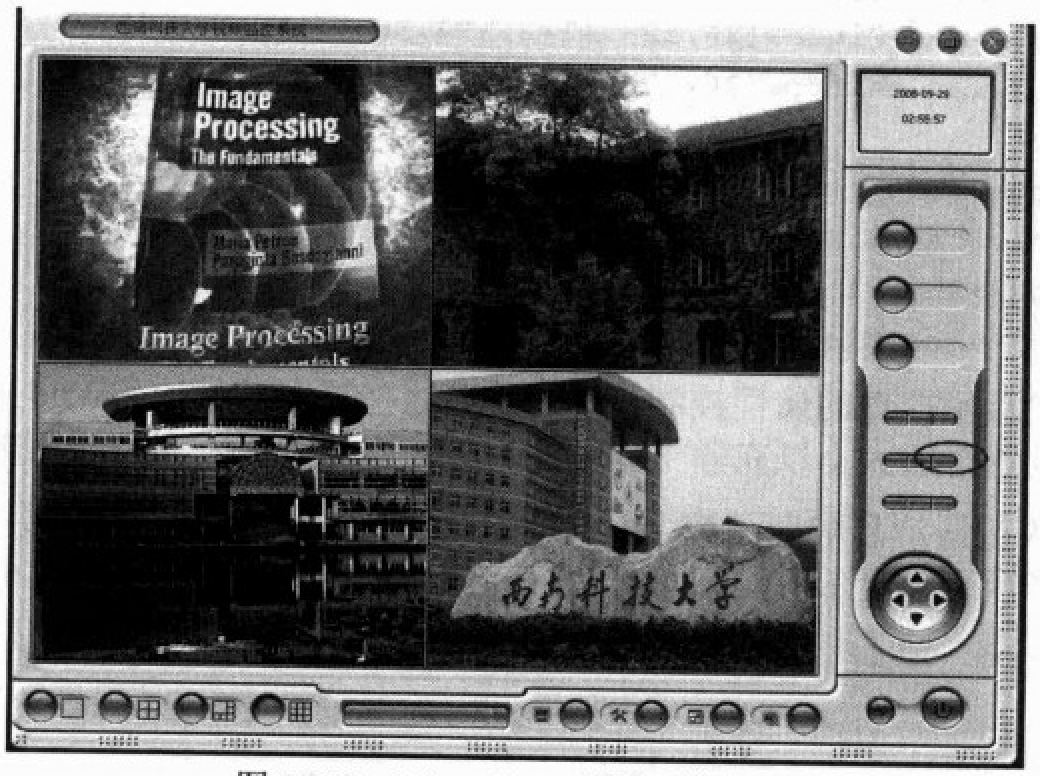


图 16-13　DirectShow 图像采集显示

视频监控系统的音/视频数据来自于视频服务器，因此编/解码后的影音数据在监控端显示、图像处理等前，首先必须解码。下面就展现使用第 8 章的基于 DirectShow 的媒体回放案例，在监控视频主程序中添加影音文件的回放功能。

16.3.2　基于DirectShow的影音文件回放

视频监控系统中心对监控现场的控制是通过对采集到的压缩音/视频流进行解码，然后根据图像内容作出判断或控制。多画面实时解码、显示是监控中心软件的核心工作。下面展现如何使用 DirectShow 回放多个影音文件。

基于项目 vss 重新复制一个项目 vss_1，对项目 vss_1 添加多个影音文件的回放功能。

1．配置 DirectShow 的开发环境

在 VC++ 2005 的开发环境中，配置 DirectShow 的有关头文件和库文件路径，详见第 7 章。把第 8 章中案例 8 的影音文件回放类 CDXGraph.cpp 和 CDXGraph.h 复制到当前项目目录下。然后把这两个文件添加到项目中，并添加如下代码。

```
#include "CDXGraph.h"    //ljz
#define CHAN_SUM 4
```

然后在类 CMainFrame 中定义多个成员函数和成员变量。

```
//play
CDXGraph *m_pFilterGraph[CHAN_SUM];   //文件回放类
CString m_sourceFile[CHAN_SUM];       //回放文件路径名称
CString m_mediaFileName[CHAN_SUM];    //回放的文件名
void CreateGraph(int chan);           //创建滤波器图表管理器
void DestroyGraph(int chan);          //销毁滤波器图表管理器
void OpenFilePlay(int chan);          //打开、回放文件
int iCurChan;                         //当前通道所引号
```

上述代码是影音回放的应用层为实现文件播放而定义的成员变量和成员函数。目前定义了 4 路回放，而实现了 3 路回放的功能。

2. 定义功能模块

针对输入的影音文件创建 DirectShow 回放文件所必需的滤波器图表，设置显示窗口。销毁滤波器管理图表在退出或重新打开文件回放时调用。

```
void CMainFrame::CreateGraph(int chan)
{
 DestroyGraph(chan);                          //销毁滤波器链表图
 m_pFilterGraph[chan] = new CDXGraph();       //创建CDXGraph对象
 if (m_pFilterGraph[chan]->Create())          //创建滤波器链表管理器
 {
            //获取输入的文件路径及名称
        TCHAR *ch1 = m_sourceFile[chan].GetBuffer(m_sourceFile[chan].GetLength());
        if (!m_pFilterGraph[chan]->RenderFile(ch1))//渲染媒体文件，构建滤波器链表
        {
            MessageBox(_T("无法渲染此媒体文件！请确认是否安装相关解码器插件！\n 或者此
媒体文件已损坏！"),_T("系统提示"),MB_ICONWARNING);
            return;
        }
        m_sourceFile[chan].ReleaseBuffer();
            //设置图像显示窗口
        m_pFilterGraph[chan]->SetDisplayWindow(pStaticVideo[chan]->GetSafeHwnd());
            //设置窗口消息通知
        m_pFilterGraph[chan]->SetNotifyWindow(this->GetSafeHwnd());
        //显示第一帧图像
        m_pFilterGraph[chan]->Pause();
 }
}
void CMainFrame::DestroyGraph(int chan)
{
 if (m_pFilterGraph[chan] != NULL)
 {
        m_pFilterGraph[chan]->Stop();     //停止播放
        m_pFilterGraph[chan]->SetNotifyWindow(NULL);
```

```
        delete m_pFilterGraph[chan];      //删除指针
        m_pFilterGraph[chan] = NULL;
    }
}
```

上述代码是应用层为使用包含DirectShow功能的类CDXGraph而建立的创建和销毁功能函数。首先函数 Create()创建滤波器链表，然后渲染媒体文件，链接滤波器链表。接着设置视频显示窗口和消息通知窗口，最后暂停媒体文件而显示在第一帧图像。

打开指定通道并回放文件的功能实现如下。

```
void CMainFrame::OpenFilePlay(int chan)
{
 CString strFilter = _T("AVI File (*.avi) | *.avi|");
 strFilter += "MPEG File (*.mpg; *.mpeg) | *.mpg; *.mpeg|";
 strFilter += "MP3 File (*.mp3) | *.mp3|";
 strFilter += "WMA File (*.wma) | *.wma|";
 strFilter += "All File (*.*) | *.*|";
 CFileDialog dlg(TRUE, NULL, NULL, OFN_PATHMUSTEXIST|OFN_HIDEREADONLY, strFilter,
this);
 if (dlg.DoModal() == IDOK)
 {
      m_sourceFile[chan] = dlg.GetPathName();              //获取预回放的文件路径
      this->pStaticVideo[chan]->ShowWindow(SW_SHOWNORMAL);//显示图像窗口
      CreateGraph(chan);                                   //创建图表管理器
      m_pFilterGraph[chan]->Run();                         //播放影音文件
 }
}
```

上述代码实现选择、打开媒体文件，根据通道号正常显示视频窗口，然后创建、链接滤波器链表。

3. 回放消息处理函数

影音文件在回放时，暂停、停止等状态时会通知主窗口一些信息以便于做出控制。这里同样使用消息通知的方式。

STEP 01 在类 CMainFrame 中添加消息函数声明。

```
afx_msg LRESULT OnGraphNotify(WPARAM inWParam, LPARAM inLParam);
```

STEP 02 在类 CMainFrame 的消息映射段中添加消息映射。

```
BEGIN_MESSAGE_MAP(CMainFrame, CBkDialogST)
 //......
 ON_MESSAGE(WM_GRAPHNOTIFY, OnGraphNotify)
END_MESSAGE_MAP()
```

STEP 03 消息通知函数的实现。

这里实现当影音文件回放结束时，继续重新开始回放。

```
    LRESULT CMainFrame::OnGraphNotify(WPARAM inWParam, LPARAM inLParam)
    {
     IMediaEventEx *pEvent = NULL;
     if ((m_pFilterGraph[iCurChan]!=NULL) && (pEvent = m_pFilterGraph[iCurChan]->
GetEventHandle()))
     {
            LONG eventCode = 0;                 //事件码
            LONG eventParam1  = 0;              //事件码的第一个参数
            LONG eventParam2 = 0;               //事件码的第二个参数
              //获取事件码及其参数
            while (SUCCEEDED(pEvent->GetEvent(&eventCode, &eventParam1, &eventParam2,
0)))
            {
                  pEvent->FreeEventParams(eventCode, eventParam1, eventParam2);
                                                //释放事件码及其参数
                  switch (eventCode)            //根据事件码的内容处理事件
                  {
                  case EC_COMPLETE:             //回放结束
                        m_pFilterGraph[iCurChan]->SetCurrentPosition(0);
                                                //从头重新开始回放
                        break;
                  case EC_USERABORT:            //用户终止
                  case EC_ERRORABORT:           //出错终止
                        break;
                  default:
                        break;
                  }
            }
     }
     return 0;
    }
```

上述代码是媒体回放的消息处理函数，前面已经注册了消息处理窗口，消息标识为 **WM_GRAPHNOTIFY**。在这里可以添加各种消息的处理函数，如影音文件回放结束时重新从头开始回放等。

4. 通道回放处理

案例支持 3 个通道同时回放，通道的单击消息处理如下。目前此案例只支持第一通道的循环播放，其他通道回放结束时需要重新打开文件。读者可自行实现其他通道的循环播放功能。

```
    void CMainFrame::OnBnClickedButtonChanL() //第一个通道
    {
     iCurChan = 0;  //当前通道号
     OpenFilePlay(0);
```

```
}
void CMainFrame::OnBnClickedButtonChanM() //第二个通道
{
 OpenFilePlay(1);
}
void CMainFrame::OnBnClickedButtonChanR() //第三个通道
{
 OpenFilePlay(2);
}
```

上述代码根据不同通道的按钮消息，以通道号作为参数调用打开并播放函数 OpenFile。

5．打开影音文件后，统一时间回放

STEP 01 添加按钮，命名为 IDC_BUTTON_PLAY_ALL，捆绑控件变量，然后手工修改为 CxSkinButton 基类形式。

STEP 02 添加两个位图资源，命名为 IDB_ALL_D 和 IDB_ALL_N。

STEP 03 获取按钮 CWnd 指针，移动到指定位置并设置大小，在函数 InitInterface 中增加：

```
    //......
pBtnPlayAll = GetDlgItem(IDC_BUTTON_PLAY_ALL);
    //......
pBtnPlayAll->MoveWindow(167,704,37,37);
    //......
```

STEP 04 添加按钮单击消息处理。双击按钮 IDC_BUTTON_PLAY_ALL，添加如下代码。

```
void CMainFrame::OnBnClickedButtonPlayAll()
{
 //TODO: 在此添加控件通知处理程序代码
 int i;
 for (i=0; i<4; i++)
 {
      if (m_pFilterGraph[i])
      {
            m_pFilterGraph[i]->Run();
      }
 }
}
```

上述代码把已经创建成功的通道回放其对应的影音文件。

6．系统退出时，释放前期创建的资源

把前期创建的资源包括滤波器链表等释放。销毁函数 DestroyGraph()先停止回放，然后消息通知函数清空，最后把回放对象指针清空。

```
void CMainFrame::OnBnClickedButtonExitApp()
{
 //TODO: 在此添加控件通知处理程序代码
 //释放资源
 if (MessageBox(_T("确定要退出该账户吗？"),
                    _T("退出系统"),MB_OKCANCEL|MB_ICONQUESTION)==IDCANCEL)
        return ;
 else
 {
        KillTimer(0);
        if (bChanged)
        {
               DEVMODE   lpDevMode;
               lpDevMode.dmBitsPerPel   = 32;
               lpDevMode.dmPelsWidth    = ScrWidth;
               lpDevMode.dmPelsHeight   = ScrHeight;
               lpDevMode.dmDisplayFrequency=60;
               lpDevMode.dmSize = sizeof(lpDevMode);
               lpDevMode.dmFields = DM_PELSWIDTH|DM_PELSHEIGHT|DM_BITSPERPEL;
               LONG result = ChangeDisplaySettings(&lpDevMode,0);
               if(result == DISP_CHANGE_SUCCESSFUL) {
                      ChangeDisplaySettings(&lpDevMode,CDS_UPDATEREGISTRY);
                      bChanged = TRUE;
               } else  {
                      ChangeDisplaySettings(NULL,0);
               }
        }
        int i;
        for (i=0; i<4; i++)
               DestroyGraph(i);

        SendMessage(WM_SYSCOMMAND, SC_CLOSE, NULL);
 }
}
```

上述代码是主界面中的退出按钮的消息处理函数，主要原则是销毁、释放申请的指针和内存等。这里实现的是对 4 个通道释放滤波器链表。

16.4 系统效果展示

前面给出了系统部分功能的运行结果，图 16-14 是监控系统的所有功能集成后的运行效果图。其中遇到一个问题，第一次打开的文件在回放时，没有图像显示，但是系统确实在解码，这是由 DirectShow 在回放多个媒体文件时，系统资源不够造成的。解决办法有两个：一

是升级系统硬件资源，二是重载消息处理函数 OnEraseBkgnd。

图 16-14　视频监控系统多文件回放

16.5　本章小结

本章介绍了 DVR 视频监控的分类和功能组成，主要有两大类：PCI 式 DVR 和嵌入式 DVR。一般的监控系统包括监控现场、视频服务器、网络传输和监控中心几大功能部分，视频服务器是核心，监控中心是设计友好的系统软件。视频监控的主界面一般都是设计漂亮而简洁的，所以需要位图按钮来实现。以一个实际的案例分析了位图按钮的设计及其技术开发过程。最后通过 DirectShow 的视频采集显示模拟 DVR，多个文件的解码回放显示展示了 DirectShow 播放多文件的技术过程。本案例可以直接使用，读者也可以在此基础上继续丰富和完善，设计出功能更为强大的视频监控中心软件。

附录 A

旧版 VC 升级到 VC 2005 程序安全的 10 点注意事项

- 不要认为 strcpy_s 和 strncpy_s（及其他的字符串函数）在空间不够的时候会自动终止复制。如果需要自动截断，请使用 strncpy_s。
- 记住 fopen_s 默认是独占模式。如需共享使用文件，应该使用_sopen。
- 别忘了_dupenv_s，它将比_getenv_s 更容易使用，因为它能自动分配一个正确长度的内存。
- 在 scanf_s 中小心参数顺序。
- 确定 printf_s 中格式字符串的正确。
- 使用_countof（x）来取代 sizeof（x）/sizeof（element）。_countof 将会正确地计算元素个数，而且如果 x 是一个指针，编译器将会发出一个警告（来提醒程序员，仅针对 C++编译）。
- 记住所有的 sizes（大小，非长度）都是使用 characters（字符，unicode 下一个字符占 2 个 Byte）作为单位，而不是 Bytes（字节）。
- 记住所有的 sizes（大小，非长度，缘由同上）包含了字符串结束符'\0'（即别忘了很多情况下 size 需要+1）。
- 调试的时候监视数据 0xfd。（在调试版本下）0xfd 将会被填充在数据（buffer，通常是字符串）的结尾处。如果运行非你所愿，可能会得到一个长度错误。
- 检查所有的错误。许多新函数相比旧函数，能返回（表示）错误信息（的数值）。

A 实验

附录B

开发常见问题

- 在重载虚拟函数时，出现“某个类为只读而无法添加”的错误提示，如图 B-1 所示。解决方法是：删除对应的.ncb 文件，NCB（No Compile Browser）存放了供 Class View、Wizard Bar 和 Component Gallery 使用的信息，由 VC 开发环境自动生成，工程多次修改、编译都会生成新的信息以适应新的环境变量，这样就造成 NCB 文件变大、函数变量等不能联想等问题。所以每一次都可以删掉该文件。

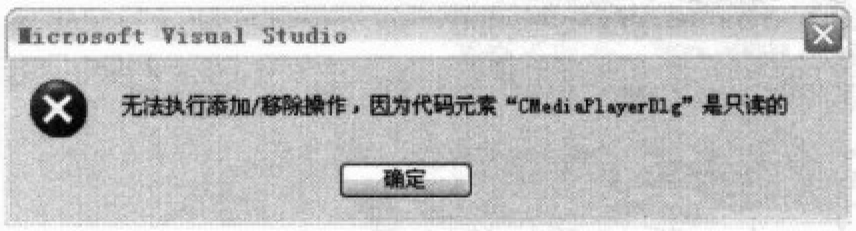


图 B-1　错误提示

- 没有找到 mfc80d.dll 的解决方法。

问题出现在程序运行清单上，默认是“嵌入清单”，清单文件是“$(intdir)\$(targetfilename) embed.manifest”。

在调试程序运行时，不知道为什么定位不到这个文件，我们如果手动把“程序名.embed.manifest”改为“程序名.manifest”，调试程序即可定位到。

所以，需要更改默认项目的工程属性。工程属性→配置属性→清单工具→输入和输出→嵌入清单→改为否。

- 在编辑状态下发现类的成员变量或函数不能显示，删除该项目扩展名为.ncb 文件，重新打开该项目。

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为；歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：（010）88254396；（010）88258888
传　　真：（010）88254397
E-mail:　dbqq@phei.com.cn
通信地址：北京市万寿路173信箱
　　　　　电子工业出版社总编办公室
邮　　编：100036